Prüfer:	Prof. Dr. Rothermel
Betreuer:	Dipl.-Inform. Fritz Hohl
Beginn am:	08.10.1996
Beendet am:	07.04.1997
CR-Nummern:	C.2.4, H.3.5

Diplomarbeit-Nr. 1470

Ein verteiltes Serversystem für die Codemigration mobiler Agenten

Peter Klar

Kurzfassung

Mobile Agenten sind Objekte, die autonom in einem Agentensystem arbeiten und aus eigenem Antrieb von einem Agentensystem zu einem anderen Agentensystem migrieren können. Bei der Migration werden die Objektdaten und der benötigte Programmcode an den Zielort übertragen. Für die Performanz eines Agentensystems ist der Zeitbedarf bei der Migration ein wichtiger Einflußfaktor. Während die Migration der Agentendaten nicht wesentlich beschleunigt werden kann, sind bei der Codemigration Verbesserungen möglich.

Ein Codeserver speichert Programmcode und stellt ihn auf Anfrage zur Verfügung. Das Problem der Codemigration kann vereinfacht auf die Suche nach einem günstigen Codeserver zurückgeführt werden. Das Problem wurde in zwei unterschiedliche Aufgaben aufgeteilt. Ein Basismechanismus garantiert das korrekte Ergebnis der Suche, während ein schneller Dienst nur manche Klassen auffindet, dafür aber besonders schnell arbeitet. Für den Basismechanismus werden eine Reihe von alternativen Algorithmen untersucht und verglichen. Der schnelle Dienst setzt sich aus verschiedenen Konzepten der Kooperation von Servern und dem Agentensystem zusammen. Die Kombination beider Suchmechanismen sorgt für eine zuverlässige und schnelle Codemigration.

Das Agentensystem muß die Sicherheit der Agenten gewährleisten, da manipulierte Agenten im Namen ihres Benutzers großen Schaden anrichten können. Das Agentensystem ist dabei darauf angewiesen, daß die Codemigration gegen Manipulationen geschützt wird. Ein Mechanismus zur automatischen Signierung der Klassen ist Teil der Implementierung des Codeservers.

Inhaltsverzeichnis

1 Einführung....1
1.1 Motivation und Aufgabenstellung....1
1.2 Mobile Agenten....3
1.3 Das Agentensystem »Mole«....3
1.3.1 Agenten, mobile Agenten....3
1.3.2 Orte, Lokationen....4
1.3.3 Engines....4
1.3.4 Migration....5
1.3.5 Klassenserver, Codeserver....6
2 Problemanalyse....9
2.1 Die externe Sicht des Serversystems....9
2.1.1 Grundlegende Funktionsweise....9
2.1.2 Eigenschaften der Codebasis....11
2.1.3 Schnittstellen....12
2.1.4 Eigenschaften des Codes mobiler Agenten....15
2.2 Restriktionen....17
2.2.1 Speicherplatz....17
2.2.2 Ortsverzeichnis....17
2.2.3 Aufkommen an Klassencode....18
2.2.4 Netzwerk....19
2.3 Interne Sicht des Serversystems....19
3 Zusammenarbeit mit Mole....21
3.1 Anbindung des Servers an Mole....21
3.1.1 Anbindung an Lokationen....22
3.1.2 Agenten als Klassenserver....23
3.1.3 Lokale Anbindung an die Engine....24
3.1.4 Eigenständiger Codeserver....25
3.1.5 Lokale und entfernte Anbindung an die Engine....29
3.1.6 Vergleich der Möglichkeiten....30
3.2 Anbindung an Java....32
3.2.1 Laden aus dem Dateisystem....32
3.2.2 Indirektes Laden von Klassen....33
4 Der Basismechanismus....35
4.1 Bestätigtes Propagieren....36
4.2 Multicast....37
4.3 Beheimatete Klassen....38
4.3.1 Der Mechanismus....38
4.3.2 Replikate....39

4.4 Verwenden des »Domain Name Service«.....41
4.4.1 Komplette Namensauflösung durch den DNS.....41
4.4.2 Beheimatete Klassen und DNS.....42
4.5 Verteilter Multibaum.....44
4.5.1 Die Grundstruktur des verteilten Multibaums.....45
4.5.2 Suchen im verteilten Multibaum.....46
4.5.3 Aufnehmen neuer Klassen.....47
4.5.4 Vergabereihenfolge der Servernummern.....47
4.5.5 Anmelden eines neuen Servers.....49
4.5.6 Abmelden eines Servers.....51
4.5.7 Einführen von Replikaten.....51
4.6 Vergleich der Basismechanismen.....52
4.6.1 Skalierbarkeit.....52
4.6.2 Netzbelastung.....53
4.6.3 Antwortzeit.....54
4.6.4 Fehlerverhalten.....55
4.6.5 Lastbalancierung.....55
4.6.6 Ressourcenverbrauch.....56
4.6.7 Einschränkungen.....56
4.7 Implementierter Mechanismus.....57
5 Verbesserung der Performanz.....59
5.1 Der Zwischenspeicher.....60
5.1.1 LRU (least recently used).....63
5.1.2 Random-Algorithmus.....66
5.1.3 Vergleich der Bewertungsfunktionen.....67
5.2 Schnelle Verbindungen suchen.....71
5.2.1 Nachbarserver.....71
5.2.2 Besorgungshinweise.....76
5.3 Vorratshaltung.....77
5.4 Suchreihenfolge.....80
6 Sicherheit.....83
6.1 Prinzip und Zielsetzung.....83
6.2 Sichere Schlüssel.....85
6.3 Sicherheit durch digitale Signaturen.....86
6.4 Versionsverwaltung.....87
7 Schnittstellenbeschreibung.....89
7.1 Schnittstelle zum Programmierer.....89
7.2 Schnittstelle zum Agentensystem.....91
7.3 Schnittstelle innerhalb des Serversystems.....93
8 Zusammenfassung und Ausblick.....95

9 Anhang..97
9.1 Implementierung..97
9.2 Beispielprotokolle..99

1 Einführung

In diesem Kapitel soll die Aufgabe der Diplomarbeit erklärt und motiviert werden, dazu ist auch eine kleine Einführung in mobile Agenten, die verwendeten Begriffe und deren Bedeutung im Projekt »Mole« notwendig.

1.1 Motivation und Aufgabenstellung

Die weltweite Vernetzung der Computer ist nicht neu, und dennoch erregte sie erst in den letzten Jahren die Aufmerksamkeit einer großen Zahl von Benutzern[1]. Dieser Ansturm führte weltweit zu einem starken Zunahme der Netzwerkauslastung, so daß inzwischen nur noch von Überlastung gesprochen wird.

Das gebräuchliche Client-Server Paradigma unterstützt eine automatische Informationssuche der Benutzer nicht befriedigend. Für den Benutzer gestaltet sich die Informationssuche als häufige Wiederholung der drei Aktionen: klicken, warten und lesen. Besonders ärgerlich ist für den Heimanwender, der den Zugang zum Internet über Telefonleitungen herstellt, daß während er auf die Antwort eines überlasteten Servers warten muß, der Gebührenzähler der Telefonverbindung munter weiterläuft. Der Benutzer muß nicht nur auf die Antwort warten, er muß die Wartezeit auch noch bezahlen.

Im Bestreben, die Netzwerkbelastung zu verringern, werden gegenwärtig Lösungen auf Basis mobiler Agenten untersucht. Ein Benutzer kann einen mobilen Agenten mit der Lösung einer Aufgabe betrauen, setzt den Agenten ab und wartet ohne eine stehende Verbindung zum Netzwerk. Nur für das Abholen des Ergebnisses muß wieder eine Verbindung hergestellt werden. Weitere Vorteile von mobilen Agenten ist [Harrison et al. 1995] zu entnehmen.

1 Selbstverständlich gibt es nicht nur männliche Benutzer, sondern auch Benutzerinnen. Da bislang noch keine befriedigende Lösung dieses sprachlichen Problems gefunden werden konnte (siehe Vorschläge von Deininger et al. in »Ein Leitfaden zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom- und Doktorarbeiten am Beispiel Informatik«.), steht in dieser Arbeit die maskuline Form sowohl für männliche wie auch weibliche Personen.

Ein mobiler Agent kann zur Lösung seiner Aufgabe seinen Aufenthaltsort innerhalb des Netzwerks wechseln. Bei der Migration eines mobilen Agenten soll er am Zielort so wiederhergestellt werden, wie er den letzten Aufenthaltsort verlassen hat. Dafür müssen die Daten des Agenten, seine Ausführungspunkte und sein Code an den Zielort gebracht werden.

Im Rahmen dieser Diplomarbeit soll die Codemigration näher untersucht werden. Ziel der Codemigration ist, daß der Programmcode eines Agenten effizient am Zielort bereitgestellt wird.

Die Lösung des Problems scheint bereits bekannt zu sein, sind doch einige Server in Agentensystemen schon in der Lage Java-Code auszuliefern, zum Beispiel der Code Server in »Mole« oder der Hall Server im Projekt JAVA-TO-GO [JavaToGo 1997]. In beiden Lösungen wird fehlender Klassencode von dem Ort angefordert, den der Agent zuletzt besucht hat. Zwei Aspekte sind bislang jedoch weitgehend unbeachtet oder nur wenig befriedigend gelöst:

1. Die Performanz
   Bislang wandert der Datenzustand eines Agent und der Agentencode parallel entlang den gleichen Migrationspfaden. Am neuen Aufenthaltsort werden benötigte Klassen auf Anfrage vom alten Aufenthaltsort nachgefordert. Dieses Vorgehen ist in zweierlei Hinsicht verbesserungsfähig: Warum sollten die Klassen immer erst auf Verlangen («on-demand«) besorgt werden? Warum sollte die Klasse immer vom letzten Aufenthaltsort besorgt werden, wenn die Klasse möglicherweise von einem anderen Ort schneller besorgt werden kann?
2. Klassenreferenzen zur Laufzeit
   In der Sprache Java ist es erlaubt Objekte von Klassen, deren Klassennamen erst zur Laufzeit berechnet wurde, zu erzeugen. Ein Beispiel soll die Möglichkeiten verdeutlichen, die damit verbunden sind: Ein Agent verhandelt mit anderen Agenten und verwendet dazu ein Protokoll, das in einer Klasse implementiert ist. Um mit möglichst vielen Agenten möglichst weitreichend verhandeln zu können, soll kein festes Protokoll eingebaut werden. Der Agent bekommt von seinem Verhandlungspartner eine Protokollversion genannt, die für die Verhandlung verwendet werden soll. Diese Protokollversion kann der Agent in einen Klassennamen umsetzen, ein Objekt dieser Klasse würde der gewünschten Verhandlungsstrategie und dem gewünschten Protokoll entsprechen. Auf diese Weise könnte ein Agent Protokolle verwenden, die es zum Startzeitpunkt seiner Reise noch nicht gab. Er ist dadurch noch während seiner Reise erweiterbar.
   Ein Agent kann also Klassen benötigen, deren Namen erst während der Ausführung des Agenten bekannt werden. Es sollte also einen Mechanismus geben, der Klassen aufgrund ihres Namens auffindet.

Beide Aspekte sind für den Einsatz mobiler Agenten wichtig, in einem Fall wird die Geschwindigkeit des System und die Belastung des Netzwerks verbessert, im anderen Fall erhält der Agent mehr Möglichkeiten.

1.2 Mobile Agenten

Eine einheitliche Definition des Agentenbegriffs ist kaum möglich, da der Begriff in verschiedenen Fachrichtungen unterschiedlich verwandt wird. Informell ist ein Agenten »Jemand, der im Auftrag für einen anderen eine Aufgabe erledigt« [Fünfrocken 1996]. Ein mobiler Agent ist folglich ein Agent, der sich im Netz bewegen kann. Die Definition wird präziser, wenn der Begriff »jemand« einschränkt wird: nach [Straßer et al. 1996] ist ein Agent »eine Art Programm«. Damit wird deutlich, daß ein Agent ein aktives Objekt ist, das nicht nur aus Daten, sondern auch aus Programmcode besteht. Der Agent kann sich jedoch nicht uneingeschränkt im Netzwerk bewegen, da er von einer Ausführungsumgebung, dem Agentensystem, abhängig ist.

Im Rahmen des Projekts »Mole«, der Abteilung Verteilte Systeme, wird an der Universität Stuttgart ein solches Agentensystem erstellt und erforscht. Soweit nicht weiter darauf hingewiesen wird, nimmt diese Arbeit immer Bezug auf das Agentensystem »Mole«.

Das Agentensystem und die Agenten sind in der Sprache Java implementiert. Die Sprache bietet alle wichtigen Elemente, die für das Agentensystem nötig sind [Hohl 1995]. Java ist eine klassenbasierte, objektorientierte Programmiersprache [JavaSpec 1997], die als maschinen- und plattformunabhängige Sprache entworfen wurde.

1.3 Das Agentensystem »Mole«

In diesem Abschnitt werden der grundlegende Aufbau des Agentensystems »Mole« beschrieben und es werden einige Begriffe erklärt, soweit diese für das Verständnis der Arbeit wichtig sind.

1.3.1 Agenten, mobile Agenten

Mobile Agenten sind Objekte in der Laufzeitumgebung von Java. Jedes Objekt ist eine Instanz einer Klasse, welche die Objekteigenschaften definiert. Die Klasse von mobilen Agenten erbt stets die Eigenschaften der Klasse `mole.UserAgent`, die bereits eine Unterklasse von `mole.Agent` ist. Jeder mobile Agent besitzt einen Namen, der verschieden ist von den Namen aller anderen mobilen Agenten.

Allgemein besteht ein mobiler Agent aus den folgenden drei Komponenten:

1. Objektdaten
   Jeder Agent besitzt eine Menge von Daten, die nur seiner Instanz zugeordnet sind. Die Daten können aus primitiven Werten und aus beliebigen Objekten bestehen. Diese enthaltenen Objekte können wiederum weitere Objekte referenzieren. Bildet man die transitive Hülle bezüglich dieser Objektreferenzen, dann erhält man die Enthält-Relation des Objekts (siehe [Klar 1996]).
2. Klassencode
   Der Klassencode eines mobilen Agenten ist jeglicher Code, der von der `start`-Methode des Agenten aus erreichbar ist. Der erreichbare Code wird für die Ausführung des Agenten benötigt.
3. Ausführungspunkte
   Der Agent wird bei seiner Erzeugung gestartet, indem seine `start`-Methode aufgerufen wird. Dem Agent ist es möglich neue Ausführungspunkte zu erzeugen, um zum Beispiel asynchrone Kommunikation mit einem anderen Agenten durchführen zu können.

Systemagenten sind keine mobilen Agenten, weil sie aus Gründen der Systemsicherheit nicht migrieren können. »Agent« ist der Oberbegriff für mobile Agenten und Systemagenten.

1.3.2 Orte, Lokationen

Das Agentensystem stellt den mobilen Agenten Orte zur Verfügung an denen sie sich aufhalten, miteinander kommunizieren und Dienste erbringen beziehungsweise nützen können. Diese Aufenthaltsorte werden in Mole als Lokationen (locations) bezeichnet. Die Lokation stellt den mobilen Agenten zahlreiche Ressourcen zur Verfügung. Insbesondere sorgt die Lokation für die Ausführung, das Beenden und das Migrieren von mobilen Agenten.

Jede Lokation besitzt einen Namen, der im Domain Name Service (DNS) eingetragen ist. Diesen Namen der Lokation kann ein Agent als Ziel seiner Migration angeben. Eine Lokation ist auch im physischen Sinne ein Ort, das heißt eine Lokation ist nicht auf mehrere Rechner verteilt[2].

1.3.3 Engines

Eine Engine faßt eine oder mehrere Lokationen zusammen. Viele Dienste, die eine Lokation ihren Agenten anbietet, werden von der Engine erbracht. Dadurch verringert sich der

2 Replizierte Lokationen bilden scheinbar eine Ausnahme, dies trifft jedoch nicht zu, da sich ein Agent nur eine Instanz einer replizierten Lokation aussucht. Replizierte Lokationen sind in diesem Sinne vielmehr eine Sammlung von Lokationen, die von Agenten unter dem gleichen Namen angesprochen werden.

Aufwand für eine Lokation beträchtlich, und es kann auch eine große Zahl von Lokationen effizient verwaltet werden.

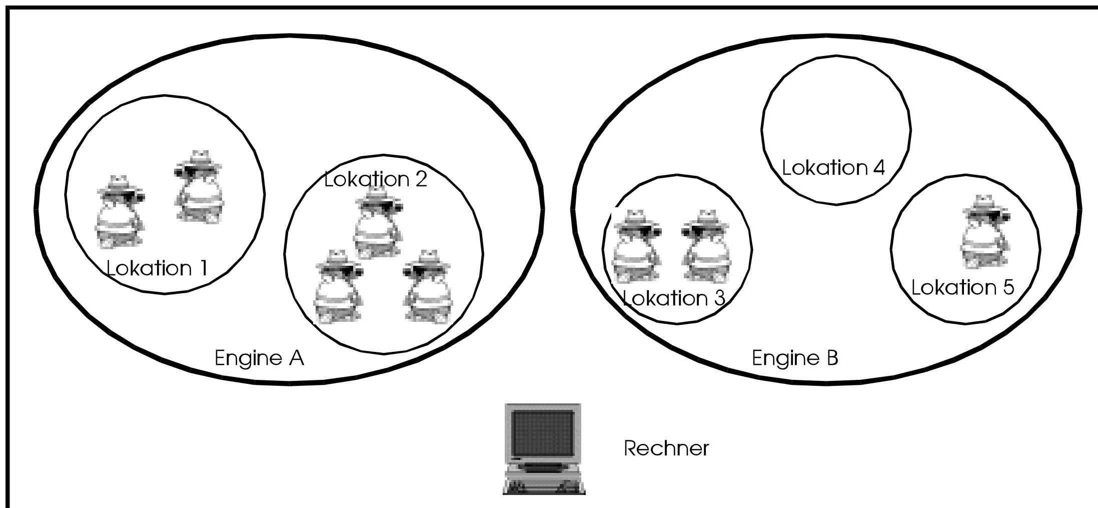

Abbildung 1: Ein Rechner mit den beiden Engines A und B. Die Engine A verwaltet zwei Lokationen, die Engine B sogar drei.

Auf einem Rechner können durchaus mehrere Engines laufen, allerdings läuft jede Engine in einem eigenen Prozeß des Betriebssystems. Um vom Betriebssystem mehr Rechenleistung zugewiesen zu bekommen, ist es durchaus praktisch mehrere Engines in separaten Prozessen zu starten.

Auf einem Rechner können also mehrere Engines laufen, jede Engine verwaltet mehrere Lokationen und an jeder Lokation können sich mehrere mobile Agenten aufhalten, die Abbildung 1 verdeutlicht diese Beziehungen.

1.3.4 Migration

Ein mobiler Agent kann selbst bestimmen, wann und wohin er migrieren möchte. Entscheidet sich der mobile Agent für einen Ortswechsel, dann ruft er die Methode `migrateTo` der Lokation auf. Nach Abschluß der Migration findet sich der mobile Agent entweder an der gewählten Ziellokation wieder oder die Migration scheiterte.

Das Agentensystem Mole transferiert allerdings nicht den ganzen mobilen Agenten (siehe oben) an die neue Lokation. Dies betrifft alle drei Komponenten in unterschiedlicher Weise:

1. Objektdaten
   Sollte ein mobiler Agent A eine Referenz auf einen anderen Agenten B besitzen, so müßten bei der Migration beide Agenten migrieren, da sich der Agent B in den Objektdaten des Agenten A befindet. Dadurch würde jedoch die Autonomie des Agenten B verletzt

werden. Analog dürfen auch Teile des Agentensystems nicht mitgenommen werden. Diese sinnvolle Einschränkungen werden durch das Agentensystem überwacht.

2. Klassencode
   Die Menge des benötigten Klassencodes kann nicht vor der Ausführung einer Klasse bereitgestellt werden, da diese Menge zu einem festen Zeitpunkt nicht berechenbar ist. Schließlich kann die Menge beliebige Klassen enthalten, deren Namen der Agent erst zur Laufzeit erzeugt hat. Nachladen zur Laufzeit löst dieses Problem, verzögert allerdings die Ausführung der Klasse. Die möglichst effiziente Codemigration wird in dieser Arbeit noch ausführlich untersucht.
3. Ausführungspunkte
   Das Migrieren von Ausführungspunkten ist besonders kritisch, deshalb migrieren die Ausführungspunkte im Agentensystem Mole nicht. Bei der Migration von Ausführungspunkten sind folgende Fragen noch ungeklärt: wie wird eine Migration angestoßen, werden alle Ausführungspunkte migriert, wie erfahren die Ausführungspunkte von der Migration. Die bestehende Implementierung beendet bei der Migration alle Ausführungspunkte eines Agenten. Am Zielort wird ein neuer Ausführungspunkt generiert, der in der `start`-Methode aufsetzt.

Für mobile Agenten spielt die Migration eine zentrale Rolle. Im Agentensystem sind einige Komponenten bei der Migration beteiligt, der Klassenserver ist eine der Komponenten, die für die Codemigration zuständig ist.

1.3.5 Klassenserver, Codeserver

Der Klassenserver ist eine Komponente, die zu einem gegebenen Klassenamen die zugehörige Klasse besorgt. Ein Java Compiler erzeugt ein Class-File, das neben dem Klassencode im engeren Sinne noch wichtige Laufzeitinformationen enthält. Da es nicht sinnvoll ist den Klassencode und die Laufzeitinformationen getrennt voneinander zu betrachten, sind die Begriffe »Klasse« und »Klassencode« sowie »Klassenserver« und »Codeserver« hier als Synonyme zu betrachten.

In der vorliegenden Implementierung von Mole (Release 1.0 [Mole 1996]) wird bei der Übermittlung der Objektdaten auch der Name der zugehörigen Klasse übergeben, hier: Sales.class (siehe Abbildung 2, Schritt 0). Dieser Klassenname wird an den Klassenserver weitergeleitet, welcher logisch der Engine zugeordnet ist. Ist der Klassencode bereits lokal vorhanden (Schritt 1), so wird er vom Java Laufzeitsystem automatisch geladen. Ist der Klassencode noch nicht lokal vorhanden (Schritt 2), dann fordert der Klassenserver, unter Verwendung des HyperText Transfer Protocols (HTTP), den Klassencode vom Klassenserver der Lokation an (Schritt 3), auf dem sich der mobile Agent zuletzt aufgehalten hat (Schritt 4). Der Klassencode wird im Dateisystem so abgelegt (Schritt 5), daß das Java Laufzeitsystem ihn über eine Pfadangabe der Umgebungsvariablen `CLASSPATH` finden

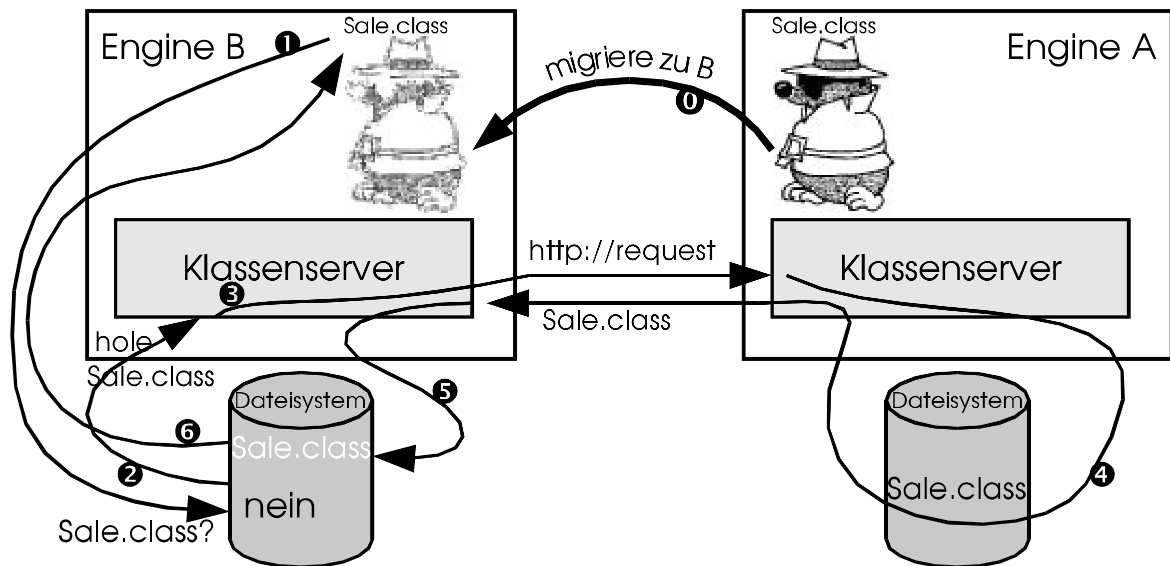

Abbildung 2: Codemigration im Agentensystem Mole

kann (Schritt 6). Nicht mehr benötigter Klassencode wird bislang nicht aus dem Dateisystem entfernt.

Nach dem Laden einer Klasse wird diese nach eventuell enthaltenen Klassennamen durchsucht. Die so gefundenen Klassennamen werden ebenfalls angefordert. Dafür werden jeweils die Schritte 1-6 der Abbildung 2 durchlaufen.

2 Problemanalyse

Die Aufgabe des Codeservers wurde bereits kurz erläutert. Für eine Untersuchung der Lösungsmöglichkeiten müssen die Anforderungen näher betrachtet werden, dabei werden nicht nur die Aufgaben, sondern auch die Randbedingungen für den Codeserver untersucht.

2.1 Die externe Sicht des Serversystems

Unabhängig von der tatsächlichen Realisierung des Serversystems können die Verhaltensweisen gegenüber der Umgebung beschrieben werden. Neben der grundlegenden Funktionsweise des Serversystems werden auch die Eigenschaften der Codebasis, der Schnittstellen und der Klassen betrachtet.

2.1.1 Grundlegende Funktionsweise

Das Serversystem soll Klassen speichern und diese auf Anfrage wieder bereitzustellen. Um den Anforderungen an Verfügbarkeit und Belastbarkeit gerecht zu werden, ist das Serversystem keine zentrale Instanz, sondern besteht aus einer großen Zahl von Servern, welche sich die Aufgabe teilen. Nach außen soll das Serversystem jedoch verteilungstransparent sein, das heißt, es hat folgende Eigenschaften:

- Zugriffstransparenz
  Der Zugriff auf lokal und entfernte Klassen geschieht in der gleichen Weise.
- Lokationstransparenz
  Der Benutzer benötigt kein Wissen über den Aufenthaltsort der Klasse. Die Zugriffs- und Lokationstransparenz bedeuten konkret, daß nur der Name der gewünschten Klasse ange-

geben werden muß, der Speicherort der Klasse interessiert den Benutzer des Serversystems nicht.

- Replikationstransparenz
  Der Zugriff auf Klassen geschieht ohne Wissen über eventuell existierende Replikate. Dies bedingt, daß Änderungen des Originals bei allen Replikaten automatisch sichtbar werden. Gleichzeitige Änderungen am Original und an einer Kopie müssen synchronisiert werden. Die damit verbundenen Probleme treten auch bei Datenbanken auf und sind zum Beispiel in [Gray et al. 1993] ausführlich erläutert.
- Fragmentierungstransparenz
  Eine eventuell vorgenommene Fragmentierung von Klassen wird nach außen hin nicht sichtbar. Wie aus den Erhebungen statistischer Daten über die Klassen in Tabelle 2 hervorgeht, ist die Fragmentierungstransparenz von geringer Bedeutung, da die Größe der Klassen so klein ist, daß eine Fragmentierung nicht notwendig ist.

Die Abbildung 3 zeigt wie das Serversystem im Prinzip funktionieren soll. An beliebiger Stelle im Netzwerk sollen einige Klassen bei dem Serversystem registriert werden. Die registrierten Klassen können dann von allen Agentensystemen im Netzwerk abgerufen werden, wobei die Agentensysteme außer dem Namen keine Informationen über die Klasse besitzen.

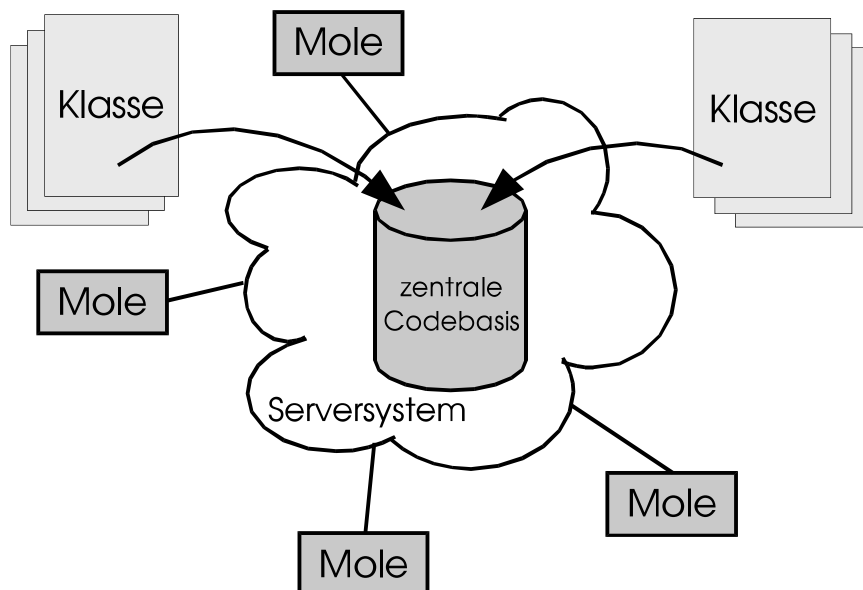

Abbildung 3: Externe Sicht des Serversystems

Von außerhalb des Serversystems kann die Codebasis als zentraler Datenbestand angesehen werden, die Eigenschaften des Datenbestands werden nachfolgend näher betrachtet.

2.1.2 Eigenschaften der Codebasis

Die Codebasis ist die Gesamtmenge an Code, die das Serversystem verwaltet. Die Codebasis kann als verteilte Datenbank betrachtet werden. Auf eine Klasse kann mit dem Klassennamen zugegriffen werden, das heißt der Klassenname ist ein Primärschlüssel. Wie bei einer Datenbank darf jeder Primärschlüssel nur einmal vorkommen, das heißt, daß zu einem Klassennamen nur eine Klasse registriert werden kann. Diese Bedingung scheint zunächst keine Probleme zu bereiten, da man den gesamten Namensraum aufteilen kann.

Wie in einer Datenbank müssen verschiedene Zugriffe auf die Codebasis isoliert werden. Insbesondere soll das erste Gesetz der Zugriffssteuerung gelten (siehe [Grey et al. 1993], S. 376ff): Zeitgleiche Programmausführung soll nicht zu Fehlverhalten von Anwendungen führen. Das heißt, daß alle Zugriffe von Anwendungen voneinander isoliert werden. Zugriffe einer Anwendung haben keinen Einfluß auf Zugriffe einer anderen Anwendung.

Im Agentensystem stellt jeder Agent eine Anwendung dar, deren Zugriffe von allen anderen Zugriffen isoliert werden muß. Da mobile Agenten nur lesend auf die Codebasis zugreifen, müssen die Zugriffe verschiedener Agenten nicht synchronisiert werden. Aber ein Agentenprogrammierer könnte eine Klasse aus der Codebasis auslesen, verändern und schließlich wieder in die Codebasis zurückschreiben. Dadurch kann es passieren, daß dem mobilen Agent in der nächsten Lokation, die er besucht, der veränderte Code zur Verfügung gestellt wird. Ein Agent muß - nach dem ersten Gesetz der Zugriffssteuerung - beim Zugriff auf eine Klasse immer dieselbe Implementierung erhalten. Eine einmal registrierte Klasse darf sich folglich erst ändern, wenn keine mobilen Agenten mehr unterwegs sind, die diese Klasse verwenden. Zu einem festen Zeitpunkt ist im allgemeinen jedoch nicht entscheidbar welche Klassen ein Agent verwendet, die Bedingung kann also erst dann sichergestellt werden, wenn überhaupt keine Agenten mehr unterwegs sind. Dies ist wiederum schwer festzustellen und zu weitreichend, daher ist an dieser Stelle ein Kompromiß notwendig.

Wenn eine Klasse erst geändert werden kann, wenn sie nicht mehr verwendet wird, dann kann ein Agentenprogrammierer seine Klasse nicht verändern, solange noch Agenten auf Basis der alten Version unterwegs sind. Als Abhilfe könne er der veränderten Klasse einen neuen Namen vergeben, dies ist für den Agentenprogrammierer jedoch nicht komfortabel, und es schleichen sich beim Programmierer leicht Fehler ein, weil dieser schnell die Übersicht über seine Versionen verliert. Eine einfache Versionsverwaltung sollte also durch den Codeserver bereits vorgegeben werden.

Ein anderes Problem der Codebasis besteht darin, daß deren referenzielle Integrität gewahrt bleiben muß. Ein Objekt einer Klasse referenziert weitere Objekte anderer Klassen. Diese Referenzen können als Fremdschlüssel bezeichnet werden, da der Klassenname ein Primärschlüssel ist. Die referenzielle Integrität [Date 1991] fordert, daß jeder Wert eines Fremdschlüssels in einer Relation R_1, einem Wert des Primärschlüssels der betroffenen Relation R_2 entsprechen muß, oder aber, daß der Wert des Fremdschlüssels ein Nullwert ist. Konkret bedeutet dies, daß nur Klassen referenziert werden dürfen, die auch in der Codebasis enthal-

ten sind. Die Bedingung kann beim Registrieren der Klasse nicht vollständig überprüft werden, da die Menge der referenzierten Klassen zu diesem Zeitpunkt nicht entscheidbar ist. Deshalb beschränkt sich die Überprüfung zunächst nur auf die *festgelegten Referenzen* einer Klasse, das sind alle Klassennamen, die schon bei der Übersetzung der Klasse angegeben wurden. Es kann allerdings Klassen geben, die zwar festgelegt-referenziert werden, aber dennoch nicht benötigt werden, umgekehrt können Klassen tatsächlich referenziert werden, für die keine festgelegte Referenz existiert. Die referenzielle Integrität kann vollständig erst zur Laufzeit sichergestellt werden, wird eine Klasse zur Laufzeit referenziert, die nicht verfügbar ist, so wird beim Referenzieren eine Ausnahmebehandlung angestoßen.

Bevor der Inhalt der Codebasis untersucht wird, sollen zunächst die notwendigen Zugriffsoperationen bestimmt werden, diese sind nach den verschiedenen Schnittstellen zu unterscheiden.

2.1.3 Schnittstellen

Zunächst ist es zweckmäßig die an den Schnittstellen aneinanderstoßenden Komponenten zu benennen. Die an der Codemigration beteiligten Komponenten sind in der Tabelle 1 aufgelistet.

Komponente	Rolle der Komponente und Wechselwirkungen
Java-Laufzeitsystem (Java Virtual Maschine)	Das Laufzeitsystem ist der »Endverbraucher« des Klassencodes. Vom Laufzeitsystem gehen Klassenanforderungen aus, wenn die entsprechende Klasse benötigt wird, um einen Ausführungspunkt fortsetzen zu können.
Agentensystem	Das Agentensystem steht bei der Codemigration als Verbindungsglied zwischen dem Codeserver und dem Laufzeitsystem. Es registriert Nachforderungen des Laufzeitsystems und leitet diese dem Codeserver weiter.
Codeserver	Der Codeserver ist Teil eines verteilten Serversystems, das die Codebasis verwaltet. Das Serversystem sorgt für die effiziente Bereitstellung der Klassen auf dem Codeserver, an dem sie nachgefragt werden.
Programmierer bzw. Codegenerator	Der Programmierer (bzw. im Fall automatischer Codeerzeugung der Codegenerator) ist der Ursprung sämtlicher Klassen. Er stellt dem Agentensystem Code zur Verfügung, indem er die entsprechende Klasse bei seinem Codeserver registriert.

Tabelle 1 : An der Codemigration beteiligte Komponenten und deren Aufgaben

Bei der Schnittstelle Laufzeitsystem - Agentensystem werden die Vorgaben vom Laufzeitsystem gemacht. In Java kann eine Klasse durch ein Objekt der Klasse »`ClassLoader`« in das Laufzeitsystem geladen werden. Wurde die Klasse in das Laufzeitsystem geladen, so

steht sie sofort zur Verfügung. Die ursprüngliche Quelle der Klasse wird dann nicht mehr benötigt. In der aktuellen Java-Version 1.0 verbleibt die geladene Klasse für immer im Hauptspeicher, für die Version 1.1 ist bereits angekündigt, daß nicht mehr benötigte Klassen aus dem Speicher entfernt werden [Java1.1 1997]. Alle Klassen, die aufgrund der geladenen Klasse benötigt werden (zum Beispiel die Superklasse oder referenzierte Objekte) werden beim ClassLoader angefordert. Dieser Aufruf des ClassLoaders durch das Laufzeitsystem wird erst bei Bedarf getätigt, das heißt eine Klasse wird vom Laufzeitsystem erst angefordert, wenn ein Objekt der Klasse erzeugt werden soll. In Java gibt es folgende Möglichkeiten ein Objekt zu erzeugen:

1. Mit dem Befehl »new«, der in der Sprachdefinition von Java festgelegt ist, kann ein neues Objekt erzeugt werden. Dazu muß die zugehörige Klasse[3] im Quelltext angegeben werden, in diesem Fall steht der Name der Klasse bereits zur Übersetzungszeit fest und kann zur Laufzeit nicht geändert werden.
2. Viel mächtiger ist die Möglichkeit Objekte einer beliebigen Klasse zu erzeugen. Dabei wird zunächst der Name der Klasse in einem String abgelegt. Mit der Klassenmethode »forName« (der Klasse »Class« siehe [JavaAPI 1996]) kann der Klassendeskriptor dieser Klasse abgefragt werden. Mit dem Klassendeskriptor ist es möglich ein neues Objekt dieser Klasse zu erzeugen. Durch die Übergabe des Klassennames in einer Stringvariable, ist es möglich den Klassennamen erst zur Laufzeit zu berechnen. Zu einem beliebigen aber festen Zeitpunkt ist es somit nicht abzusehen, welche Klasse bei der Erzeugung tatsächlich referenziert wird.
3. Importierte Objekte
   Objekte können in einer Laufzeitumgebung dadurch entstehen, daß ein Objekt an einem anderen Ort erzeugt wurde und in einen Datenstrom geschrieben wurde. Das Auslesen des Datenstroms führt zur Rückgewinnung des Objekts. Auch in diesem Fall kann erst zur Laufzeit entschieden werden, welche Klasse zur Erzeugung dieses Objekts notwendig ist.

Bei Referenzen der ersten Art handelt es sich um *festgelegte Referenzen*, die anderen Referenzen sind *Laufzeitreferenzen*. In der Regel wird der Java Befehl »new« verwendet, um ein neues Objekt zu erzeugen; Laufzeitreferenzen werden selten verwendet, bieten allerdings mehr Flexibilität.

Die Möglichkeiten, die sich dem Programmierer durch die Objekterzeugung mit Laufzeitreferenzen bieten, sind deutlich flexibler als durch die Objekterzeugung mit festgelegten Referenzen. Ein Verzicht auf die Laufzeitreferenzen würde die Möglichkeiten der Anwenderprogrammierer folglich unzumutbar einschränken. Die Menge aller referenzierten Klassen ist zu einem beliebigen aber festen Zeitpunkt nicht entscheidbar

---

3 Mit »new« können auch Arrays erzeugt werden, für diese gibt es keine Klasse. Diese Ausnahme ist nicht bedeutend, da der Basistyp des Arrays entweder primitiv ist, dann wird das Problem javaintern gelöst, oder der Basistyp ist eine beliebige Klasse, dann behandelt Java jedes Objekt des Arrays wie ein eigenständiges Objekt.

(siehe Kapitel »Eigenschaften des Codes mobiler Agenten«), in der Schnittstelle Laufzeitsystem - Agentensystem werden alle Klassenanforderungen zur Laufzeit einzeln erzeugt. Das Agentensystem erhält einen Klassennamen und besorgt dem Laufzeitsystem diese Klasse.

Das Agentensystem leitet die Anfrage an den Codeserver weiter. Die Art der Schnittstelle zwischen dem Agentensystem um dem Codeserver ist davon abhängig, ob der Codeserver als Teil des Agentensystem durch lokale Methodenaufrufe benutzt wird, oder ob der Codeserver entfernt ist und über Kommunikation angesprochen wird.

Die Schnittstelle zwischen dem Programmierer und dem Serversystem ist nicht direkt vorhanden, da der Programmierer nicht direkt mit dem Codeserver kommuniziert, sondern nur indirekt über Hilfsprogramm, sogenannte Werkzeuge. Die Schnittstelle ermöglicht folgende grundlegenden Funktionen:

1. REGISTER
   Ein Benutzer schreibt oder generiert eine neue Klasse und möchte diese im Agentensystem verwenden. Die neue Klasse wird in das Serversystem eingebracht, registriert. Die Klasse steht damit ab sofort auf jedem Codeserver zur Verfügung.
2. REMOVE
   Eine Klasse wird aus dem Serversystem entfernt, da für diese Klasse ein Nachfolger existiert, oder die Klasse nicht mehr benötigt wird. REMOVE wird verwendet, wenn der Benutzer die Klasse aus der Codebasis entfernen möchte. Bei dieser Funktion taucht wieder die Problematik der konkurrierenden Zugriffe auf die zentrale Codebasis auf. Es sollten nicht Klassen gelöscht werden, die noch verwendet werden - dies ist jedoch schwer oder gar nicht festzustellen. Dennoch belegt jede Version einer Klasse Speicherplatz auf mindestens einem Server. Der Speicherplatz ist begrenzt und neue Klassen können nicht mehr registriert werden, wenn dieser erschöpft ist. Dem Programmierer wird zugemutet zu entscheiden, welche Klasse zu löschen ist, um belegten Speicherplatz wieder freizugeben.
3. FETCH
   Zum Übersetzen seiner Klassen benötigt der Programmierer alle verwendeten Klassen. Der Programmierer muß Zugriff auf die Klassen des Serversystems besitzen, um die verwendeten Klassen in das lokale Dateisystem zu kopieren, damit der Compiler darauf zugreifen kann. Diese Arbeit kann von einem einfachen Werkzeug übernommen werden.

Alle Operationen können auf einem beliebigen lokalen System initiiert werden, die Wirkung der Operation erstreckt sich jedoch nach möglichst kurzer Zeit auf alle Server des Serversystems.

Die Funktionen REGISTER und REMOVE, die Änderungen an der Codebasis vornehmen, können die referenzielle Integrität der Codebasis verletzten. Wie dies verhindert werden kann hängt von der Verwaltung der Codebasis ab und wird später erörtert.

Um die Konsistenz der Codebasis besser gewährleisten zu können, wird bewußt auf die Möglichkeit verzichtet, Änderungen an Klassen zu erlauben. Die Sprache Java schließt bereits aus, daß sich Code selbst modifiziert. Das Ändern von Klassencode ist also weder für den Benutzer noch für das System erlaubt. Sollen Änderungen an einer Klasse durchgeführt werden, so muß die geänderte Klasse neu registriert werden, es handelt sich dann dabei um eine andere Klasse. Die alte Version der Klasse bleibt unverändert verfügbar.

2.1.4 Eigenschaften des Codes mobiler Agenten

Ein Agent ist ein Objekt, das von einer Agentenklasse erzeugt wurde. Die Agentenklassen des Agentensystems »Mole« sind in Java implementiert. Durch das Agentensystem sind einige Funktionen der Agenten vorgegeben, welche die Agentenklassen der Benutzer von der Klasse `mole.UserAgent` erbt, die bereits eine Unterklasse von `mole.Agent` ist.

Ein Agent besteht aus einer Menge von Variablen, diese können wiederum Objekte verschiedener Klassen referenzieren. Bildet man die transitive Hülle bezüglich dieser Objektreferenzen, dann erhält man die Enthält-Relation des Objekts (siehe [Klar 1996]). Das Objekt gehört einer Klasse an, die bereits von Superklassen geerbt haben kann. Die Menge dieser Klassen wird im folgenden als *Klassenmenge* eines Objekts bezeichnet. Die *transitive Klassenmenge* eines Objekts bezeichnet die Vereinigung der Klassenmengen des Objekts und aller Objekte, die Element der Enthält-Relation des Objekts sind, also alle benötigten Klassen. Die Klassenmenge eines Objekts kann sich nach der Compilierung nicht mehr ändern, die transitive Klassenmenge bezieht sich dagegen auf einen besten Betrachtungszeitpunkt. Mit der *maximalen Klassenmenge* wird die Vereinigung der transitiven Klassenmengen zu jedem möglichen Betrachtungszeitpunkt bezeichnet. Während die Klassenmenge eindeutig und die transitive Klassenmenge bestimmbar ist, kann die maximale Klassenmenge nicht berechnet werden.

Zum Erzeugen eines Agenten wird zunächst unmittelbar seine Klassenmenge benötigt. Wird der Konstruktor der Agentenklasse gestartet, so kann der Agent seine Variablen initialisieren. Die Objekte, die der Agent referenziert, *können* erzeugt werden, dabei werden die Klassen der transitiven Klassenmenge benötigt. Der Agent kann jedoch die Objekte auch erst zu einem späteren Zeitpunkt erzeugen, zum Beispiel wenn er von einem anderen Agenten die gewünschten Informationen erhält.

Obwohl es wünschenswert ist, die maximale Klassenmenge vor der Migration schon zu kennen, ist dies leider nicht möglich. *Die maximale Klassenmenge ist nicht entscheidbar*. Dies hat zwei Gründe, zum einen können Klassen referenziert werden, deren Namen erst zur Laufzeit erzeugt werden, zum zweiten kann zu einem festen Zeitpunkt im allgemeinen nicht entschieden werden, ob ein Objekt tatsächlich erzeugt wird, wenn der entsprechende Befehl zur Erzeugung in einer bedingten Anweisung oder Schleife steht. Es gibt noch eine weitere Möglichkeit, wie die transitive Klassenmenge unvorhersehbar »erweitert« werden kann.

Eine Variable kann eine Referenz auf ein Objekt einer bestimmten Klasse aufnehmen. Es ist jedoch auch möglich dieser Variablen ein Objekt einer Unterklasse der Klasse zuzuweisen. Dieses »unbekannte« Objekt kann beispielsweise als Rückgabewert einer Methode eines anderen Agenten übergeben werden.

Die transitive Klassenmengen verschiedener Agenten sind in der Regel nicht disjukt, da Agenten zum Beispiel auch Objekte von Klassen des Java-Standards verwenden können. Während die Klassen des Java-Standards als lokal vorhanden vorausgesetzt werden können, liegen die übrigen Klassen auf einem beliebigen Rechner. Selbstverständlich müssen einige Voraussetzungen für diesen Rechner gelten:

- der Rechner ist permanent am Netzwerk angeschlossen
- auf dem Rechner läuft ein Codeserver

Jede Klasse hat einen eindeutigen Namen und liegt als Datei im »Classfile Format« [JavaCFF 1996] vor. In einer kleinen Untersuchung wurde ein wichtiges Kriterium der Klassen, ihre Größe und die Zahl der festgelegten Referenzen betrachtet. Dabei wurden wahllos 773 verschiedene Klassen aus Studentenprojekten, der Java-Standardbibliothek und dem Agentensystem untersucht. Die Tabelle 2 zeigt das Ergebnis der Untersuchung.

Anzahl analysierter Klassen	**Mittlere Größe der Klassen**	**Mittlere Anzahl festgelegter Referenzen pro**
773	2577 byte	8,9

Tabelle 2 : Durchschnittliche Größe und Anzahl festgelegter Referenzen üblicher Klassen

Es handelt sich bei den analysierten Klassen nur zu einem geringen Teil um Agentenklassen, dies ist aber nicht weiter entscheidend, da Agenten in der Regel viele Objekte enthalten, die keine Agenten sind. Eine Analyse von Agentenklassen ist nicht möglich, da die Anzahl der existierenden Agentenklassen noch zu gering ist. Der obigen Analyse liegt die Überlegung zugrunde, daß sich ein Programmierer eine gewisse Arbeitsweise angeeignet hat, die er nicht ändert, nur weil er ein Agentenprogramm schreibt. Hat der Programmierer bislang viele kleine Klassen geschrieben, so wird er auch einen Agenten so strukturieren, daß viele kleine Klassen entstehen.

Aus der mittleren Größe der Klassendateien kann auf den Speicherbedarf geschlossen werden, der benötigt wird um eine bestimmte Menge an Klassen zu speichern.

2.2 Restriktionen

Um eine möglichst passende Lösung zu finden, müssen die einzuhaltenden Restriktionen ausgemacht werden. Wegen des frühen Entwicklungstandes des Agentensystem »Mole« können die Restriktionen nicht aus der Praxis gewonnen werden, vielmehr müssen sie so gewählt werden, daß dem Gedanken des Agentensystems entsprochen wird. Die Menge der existierenden Anwendungen für »Mole« ist noch gering, so daß das Serversystem möglichst flexibel sein muß, um damit einer möglichst großen Zahl zukünftiger Anwendungen gerecht zu werden.

2.2.1 Speicherplatz

Der Speicherplatz als lokale Ressource läßt sich unterteilen in Hauptspeicher und externen Speicherplatz. Der zur Verfügung stehende Hauptspeicherplatz ist durch die Hardware vorgegeben und wird vom Betriebssystem und dem Java-Laufzeitsystem verwaltet. In der Version 1.0 des Java-Laufzeitsystems wird einmal geladener Klassencode nicht mehr aus dem Hauptspeicher entfernt. Ab der Version 1.1 wird unnötiger Klassencode durch das Laufzeitsystem automatisch entfernt (angekündigt in [Java1.1 1997]).

Der externe Speicherplatz ist ebenfalls durch die Hardware vorgegeben. Allerdings ist davon auszugehen, daß die vorhandene Plattenkapazität nicht nur für den Klassencode zur Verfügung steht. Weder das Laufzeitsystem von Java noch das Betriebssystem kann eigenmächtig Dateien löschen, die nicht mehr benötigt werden, deshalb ist eine Selbstbeschränkung auf eine bestimmte Plattenkapazität nötig.

Es ist nicht möglich allgemeingültige Grenzwerte für Plattenplatz anzugeben, da diese von der Gesamtkapazität der Festplatte und der Verwaltung durch das Dateisystems abhängen. Deshalb muß jeder Server konfigurierbar gestaltet werden, so daß diese Grenze angegeben werden kann.

2.2.2 Ortsverzeichnis

Die Anzahl der ständig laufenden Mole-Lokationen ist noch gering, ein Verzeichnis aller Orte könnte ohne Probleme erstellt und aktuell gehalten werden. Das Agentensystem »Mole« steht jedoch erst am Anfang seiner Entwicklung und die Zahl der betriebenen Lokationen wird noch stark zunehmen. Ein vollständiges Verzeichnis aller Orte könnte mehrere Millionen Orte umfassen und bereits eine Größe im Bereich von Gigabyte erreichen. Da der zur Verfügung stehende Platz beschränkt ist, kann diese Datenmenge nicht auf jedem Codeserver gehalten werden. Es ist auch nicht mehr praktikabel dieses Gesamtverzeichnis auf

jedem Codeserver aktuell zu halten, das Nachrichtenaufkommen dafür wäre zu groß. Die Zahl der Server, die ein Codeserver kennt, sollte deshalb begrenzt sein.

2.2.3 Aufkommen an Klassencode

Das Aufkommen an Klassencode ist eine der wichtigsten Entwurfsgrößen; sind nur wenige Klassen zu erwarten, so können diese leicht auf allen Servern repliziert vorliegen. Mit steigender Anzahl an vorhandenen Klassen sinkt die Anzahl der Replikate pro Klasse, die gespeichert werden können.

Ein großer Vorteil von Agentensystemen ist die Flexibilität, die dadurch erreicht wird, daß nicht nur Daten, sondern ausführbarer Code mobil ist. Der Benutzer ist nicht mehr darauf angewiesen, daß eine bestimmte Anwendung auf dem Zielrechner installiert ist, der benötige Klassencode wird dem Zielrechner automatisch bereitgestellt. Es sind daher neue Vorgehensweisen der Informationsverarbeitung denkbar, die Möglichkeiten, die sich damit eröffnen, sind momentan noch nicht abzuschätzen. Bislang wird die Anfrage eines Benutzers, zum Beispiel das Ausfüllen einer Maske, in eine Suchanfrage in einer bestimmten Sprache (zum Beispiel SQL) umgesetzt. Die Suchanfrage wird als Nachricht an den Rechner geschickt, auf dem die gewünschten Informationen vermutet werden. In einem Agentensystem wird die Suchanfrage anders bearbeitet, anstatt einer Suchanfrage wird ein neuer Agent generiert und abgeschickt. Noch mehr Effizienz ist zu erwarten, wenn gleich mehrere mobile Agenten abgeschickt werden, die untereinander kooperieren. Selbstverständlich könnte den Agenten die Benutzeranfrage als Parameter mitgegeben werden, dadurch kann für alle Anfragen ein generischer Agent verwendet werden. Dieser Agent müßte jedoch die Anfrage erst noch interpretieren und wäre dadurch nicht besonders schnell, andererseits muß der generische Agent alle Möglichkeiten berücksichtigen und ist bei einfachen Anfragen umfangreicher als notwendig. Es macht also durchaus Sinn für jede Anfrage einen speziellen Agenten zu generieren, der genau die gestellt Aufgabe löst. Auch wenn das Generieren von Agenten noch nicht angewendet wird, so ist diese Möglichkeit für die Zukunft nicht auszuschließen.

Selbst wenn die benötigten Klassen stets generiert werden, so wird es sinnvollerweise eine Klassenbibliothek geben, auf die zurückgegriffen werden kann. Von Klassen solcher Bibliotheken ist anzunehmen, daß sie stark verbreitet sind und häufig benötigt werden. Die speziellen Klassen der Benutzeragenten werden dagegen nur für wenige Agenten benötigt, sie werden folglich von wenigen Orten des Agentensystems jeweils nur wenige mal (oft nur einmal) benötigt. Durch den hohen Spezialisierungsgrad der Benutzeragenten werden diese keine weite Verbreitung finden. Die Verbreitung der Klassen ist folglich unterschiedlich, so daß es verschiedene Strategien geben muß, die möglichst vielen Fällen gerecht werden.

Geht eine Abschätzung des Klassenaufkommens von der momentanen Situation aus, so ist die Zahl der Lokationen gering, die Anzahl der Agenten ist ebenfalls klein, da noch kein Agenten-Generator vorgesehen ist. Diese Abschätzung stellt keine großen Anforderungen

an das Serversystem, in Zukunft könne jedoch ein dermaßen dimensioniertes Serversystem überfordert werden. Um auf zukünftige Entwicklungen vorbereitet zu sein, wird angenommen, daß die gesamte Menge an Klassencode die Kapazität eines Codeservers weit überschreitet. Jeder Codeserver kann also nur einen kleinen Teil der gesamten Codebasis speichern.

2.2.4 Netzwerk

Ein verteiltes Serversystem bedingt selbstverständlich ein Netzwerk, über welches die Codeserver kommunizieren können. Der Codeserver verwendet Internet-Verbindungen, die von der Sprache Java angeboten werden, es stehen nur die Dienste TCP und UDP zur Verfügung. Auf das Routingverhalten kann kein Einfluß genommen werden, auch das Auswerten der Routinginformationen ist in Java nicht möglich.

Während der Großteil der Lokationen ständig im Internet präsent sind, sollen auch Lokationen berücksichtigt werden, die nur zeitweilig am Netzwerk hängen. Damit soll einem großen Vorteil des Agentensystems Rechnung getragen werden, daß ein Benutzer nicht Online auf die Anwort warten muß, sondern einen Agenten absetzen und sich dann vom Netzwerk abhängen kann. Eine solche Off-Line-Lokation kann selbstverständlich nicht mehr Klassen nachfordern, wenn sie vom Netzwerk getrennt ist. Die Lokation muß folglich schon vorsorglich mit den benötigten Klassen versorgt werden.

Das Netzwerk wird nicht als statisch, jedoch als berechenbar vorausgesetzt. Um Optimierungen durchführen zu können, müssen Antwort- und Übertragungszeiten zwischen verschiedenen Rechnern vorliegen, wie sie auch in [Voigt 1996] gemessen werden. Diese Messungen bedingen eine gewisse Determiniertheit des Netzwerks. Diese ist im Internet nicht gegeben, jedoch ist eine ausreichende Konstanz vorhanden, so daß die Messungen über einen gewissen Zeitraum ihre Gültigkeit behalten.

2.3 Interne Sicht des Serversystems

Während das Agentensystem nach einer Anfrage nur auf das Eintreffen der Klasse wartet, ist aus der Sicht des Serversystems interessant, wie die Klasse möglichst schnell gefunden werden kann.

Um den Unterschiedlichen Anforderungen gerecht zu werden, kann das Serversystem zwei unterschiedliche Dienste unterscheiden:

1. Der Basismechanismus
   Der Basismechanismus garantiert das Auffinden einer Klasse, wenn sie sich in der Codebasis befindet, oder das Anzeigen ihrer Nichtexistenz in der Codebasis. Der Basismechanismus muß also zuverlässig und robust sein, soweit möglich sollte er auch schnell arbeiten und schonend mit Ressourcen umgehen.
2. Der schnelle Dienst
   Der schnelle Dienst kann das Finden einer Klasse nicht garantieren, dafür verkürzt er den Zeitbedarf der Migration. Der Ressourcenverbrauch muß in einem sinnvollen Verhältnis zur eingesparten Zeitdauer stehen.

Der Codeserver erledigt seine Aufgaben in dem er beide Dienste verwendet. Die beiden Dienste werden, für eine übersichtlichere Architektur des Codeservers, völlig voneinander getrennt. Beide Dienste verfügen über jeweils eigene Kontingente an externem Speicherplatz.

Mögliche Basismechanismen werden in »Der Basismechanismus« beschrieben und bewertet; im Kapitel »Verbesserung der Performanz« werden mögliche Strategien des schnellen Dienstes untersucht.

3 Zusammenarbeit mit Mole

Das Serversystem hat die Aufgabe das Agentensystem mit den gewünschten Klassen zu versorgen. Ein Server muß folglich mit dem Agentensystem kommunizieren können, dies kann durch lokale Methodenaufrufe oder entfernt über Nachrichtenaustausch geschehen. In diesem Kapitel sollen die Möglichkeiten untersucht und bewertet werden.

3.1 Anbindung des Servers an Mole

Wie bereits in »Schnittstellen« beschrieben, gibt es eine Reihe von Schnittstellen, die zu definieren sind. In diesem Abschnitt soll zunächst die Schnittstelle zwischen dem Codeserver und dem Agentensystem untersucht werden. Dabei muß zuerst festgelegt werden, wie die Schnittstelle gestaltet werden soll, bevor ein Protokoll für den Datenaustausch festgelegt werden kann.

Das Agentensystem »Mole« wird hier als gegeben vorausgesetzt. Es sei an dieser Stelle nochmals bemerkt, daß sich das Agentensystem noch in der Entwicklung befindet. Die Anbindung in diesem Abschnitt bezieht sich auf die inoffizielle Vorversion der Moleversion 2.0. Die Abbildung 4 aus [Straßer et al. 1996] zeigt die Architektur des Agentensystems Mole.

Agenten können sich im Agentensystem »Mole« nur an Orten aufhalten oder zwischen diesen migrieren. Ein Ort ist eine logische Einheit, die nicht zwangsläufig physikalisch einem Ort, also einem Rechner, entsprechen muß. Einer oder mehrere Orte werden von einer Engine verwaltet. Auf einem Rechner können mehrere Engines läuft, allerdings läuft jede Engine in einem eigenen Prozeß des Betriebssystems.

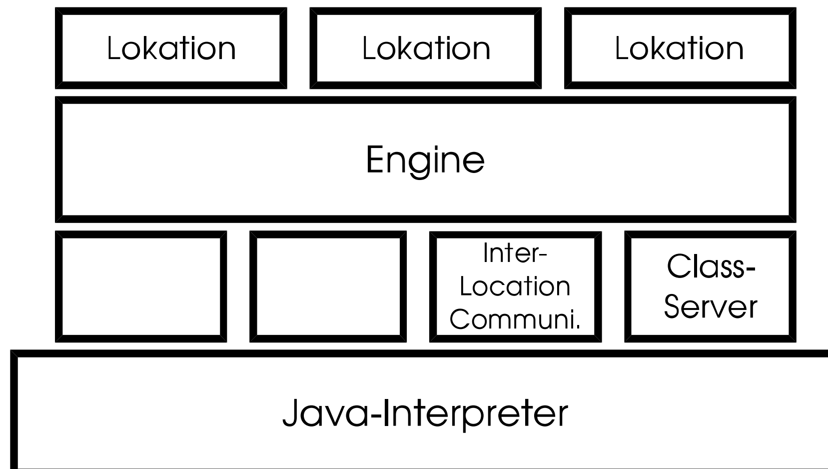

Abbildung 4: Architektur des Agentensystems Mole nach [Straßer et al. 1996]

Bislang ist der Codeserver (siehe »ClassServer« in Abbildung 4) ein lokaler Dienst der Engine. Es gibt allerdings noch andere Möglichkeiten, wie das Agentensystem mit dem Codeserver verbunden werden kann. Bei diesen Möglichkeiten wird der ClassServer aus der obigen Architektur entfernt und der Codeserver wird an der zu untersuchenden Stelle eingesetzt. Abschließend werden die verschiedenen Möglichkeiten miteinander verglichen.

3.1.1 Anbindung an Lokationen

Eine Möglichkeit der Anbindung besteht darin, jeder Lokation einen Codeserver zur Verfügung zu stellen. Die Abbildung 5 zeigt den entsprechend modifizierten Aufbau des Agentensystem.

Diese Anordnung des Codeservers bedeutet, daß der Codeserver von den Lokationen verwendet werden kann und der Codeserver wiederum Methoden der Engine verwenden kann. Jede Lokation hat Zugriff auf ihren eigenen Server, dieser Server stellt der Lokation die benötigten Klassen zur Verfügung.

Diese Anordnung der Komponenten ist nicht sinnvoll:

- Migriert ein Agent auf eine Lokation der selben Engine, dann muß der Klassencode auf dem Codeserver unter der neuen Lokation vorhanden sein. Das kann dazu führen, daß mehrere Codeserver auf einer Engine dieselben Klassen speichern. Diese Redundanz innerhalb eines Prozesses ist ohne Nutzen und damit reine Verschwendung von Ressourcen.

- Die Architektur ist nicht hierarchisch. Die Engine ist für die Migration von Agenten zwischen Lokationen verschiedener Engines zuständig, das heißt, daß der Bedarf an Klassencode auch an dieser Stelle entsteht. Aus Sicht der Engine ist es nicht günstig eine Komponente aufrufen, die auf der Engine aufbaut, dies widerspricht der Bedeutung der

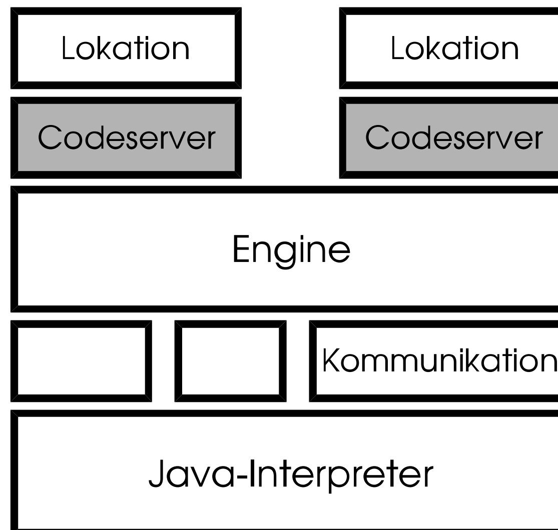

Abbildung 5: Jede Lokation kann auf einen eigenen Codeserver zurückgreifen.

Anordnung, sollte doch nur der Codeserver auf die Engine zugreifen und nicht umgekehrt. Bei einem Neuentwurf sollte eine solche Architektur vermieden werden.

Da diese Anordnung nur wenig sinnvoll ist, wird dieser Vorschlag beim Vergleich nicht mehr berücksichtigt.

3.1.2 Agenten als Klassenserver

Mobile Agenten stellen ein neues Programmierparadigma dar, an dieser Stelle ist zu prüfen, ob das Agentensystem die eigenen Probleme mittels mobiler Agenten lösen kann. Sind Agenten geeignete Codeserver, dann ist die Frage nach der Anbindung des Servers an Mole überflüssig.

Für jede Klasse wird ein Serveragent erzeugt, der für die sinnvolle Verbereitung der Klasse sorgt. Dieser Serveragent besteht selbst auch aus Code, der Serveragent kann seine eigene Klasse also nicht selbst verbreiten (diese muß schließlich schon vor ihm am Zielort sein). Der Serveragent ist folglich als Teil des Agentensystems zu betrachten, schließlich benötigt er auch das Vertrauen des Agentensystems, soll er doch neue Klassen in das System einbringen und Dateien anlegen und löschen können.

Ein Serveragent kann den Klassencode zum Beispiel verbreiten, indem er eine Kopie von sich selbst anlegt, die Kopie an den Zielort migriert. An der Ziellokation wird der Klassencode in das Laufzeitsystem gebracht, und die Kopie des Serveragenten kann sich beenden oder warten bis sie wieder gebraucht wird. Die Serveragenten können prinzipiell beliebige Strategien verfolgen.

Leider ergeben sich bei der Umsetzung mit Serveragenten mögliche Nachteile:

- Der Code des Serveragenten muß bereits bei allen Orten des Agentensystems vorhanden sein. Damit unterscheidet sich ein Serveragent nicht von einem separaten System, das nicht innerhalb des Agentensystems abläuft.
- Erst auf Ebene eines Engine besteht ein Bedarf für Klassencode. Auf einer Engine bauen (mehrere) Lokationen auf. Da Agenten sich jedoch stets an Orten aufhalten, müssen zusätzliche Mechanismen existieren, die Duplikate von Klassen in einer Engine verhindern.
- Der Serveragent besitzt weitreichende Zugriffsrechte. Es ist aufwendiger, die Sicherheit von vielen kleinen Agenten zu gewährleisten, als die Sicherheit eines Servers für alle Klassen.
- Sollte das Auffinden von Agenten für die Codemigration nötig sein, dann erfordert dies einen erheblichen Aufwand (siehe [Röhrle 1996]).

Die Umsetzung eines Serversystems für die Codemigration mit mobilen Agenten und Systemagenten ist möglich, benötigt aber zusätzliche Komponenten und erfordert einen höheren Aufwand ohne einen erkennbaren Gewinn zu erzielen. Durch die Zugehörigkeit von Agenten zu einzelnen Lokationen kommt die Realisierung durch Agenten einer Anbindung an die Lokation nahe, deshalb wird auch dieser Vorschlag beim Vergleich nicht weiter berücksichtigt.

3.1.3 Lokale Anbindung an die Engine

Um die Redundanz innerhalb einer Engine zu vermeiden, kann der Codeserver unterhalb der Engine angesiedelt werden, als ein Dienst, der von der Engine aufgerufen werden kann (siehe Abbildung 6).

Diese Anordnung entspricht der bisherigen Architektur von Mole. Die Engine führt die Datenmigration der Agenten durch, dabei entsteht ein Bedarf an fehlenden Klassen, um die Objekte der Agenten zu erzeugen. Die Engine erhält die benötigten Klassen durch Aufrufe von Methoden des Codeservers. Der Codeserver dagegen benötigt die Dienste der Engine nicht.

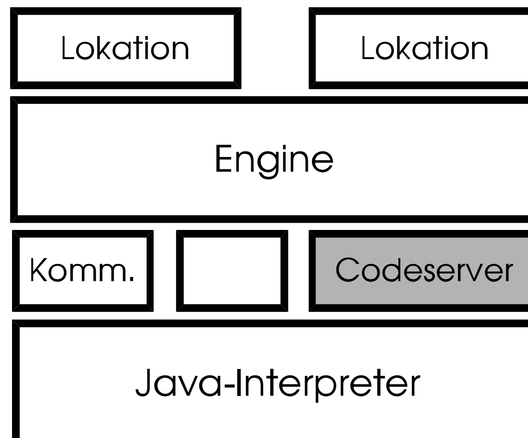

Abbildung 6: Der Codeserver als lokaler Dienst für die Engine

Es können mehrere Lokationen von einem Codeserver bedient werden, Redundanz innerhalb einer Engine ist damit eliminiert. Dennoch kann Redundanz auftreten, wenn mehrere Engines auf einer Maschine laufen, dadurch würden jedoch auch mehrere Codeserver auf einer Maschine gestartet, obgleich auch ein Codeserver ausreichen würde.

Zur Steigerung der Performanz soll der Codeserver statistische Informationen sammeln (siehe »Verbesserung der Performanz«). Je länger der Codeserver Informationen gesammelt hat, desto wertvoller können diese Informationen zur Steigerung der Performanz sein. Durch das Abbrechen des Codeservers verliert dieser seine statistischen Informationen, weil diese nicht persistent gehalten werden, denn beim nächsten Start des Codeservers können sie bereits veraltet sein. Der Verlust dieser Informationen ist für die Funktionsfähigkeit zwar unbedeutend, die Wiederherstellung der Informationen erfordert jedoch zusätzlichen Nachrichtenaustausch und bedingt längere Antwortzeiten. Obwohl Engines lang laufende Prozesse sein sollten, werden diese in der Praxis häufig abgebrochen und neu gestartet. Dies kann daran liegen, daß eine Lokation für Testzwecke nur kurz benötigt wird; die Anzahl der schon lange Zeit laufenden Agentensysteme ist äußerst gering. Das häufige Abbrechen und Neustarten der Engine ist ein Nachteil bei dieser Anbindung des Codeservers.

3.1.4 Eigenständiger Codeserver

Eine andere Möglichkeit sieht vor, daß der Codeserver nicht ein Teil des Agentensystems ist. Der Codeserver läuft in einem eigenständigen Prozeß und wird vom Agentensystem durch Nachrichten gesteuert, die Abbildung 7 zeigt diese Möglichkeit.

Für die Engine ist in dieser Architektur kein Unterschied zur lokal Anbindung an die Engine bemerkbar. Der lokale Klassenlader nimmt alle Klassenanforderungen der Engine entgegen und leitet sie an den entfernten Codeserver weiter. Der Codeserver bearbeitet die Anfrage des Klassenladers und schickt an diesen die Antwort zurück. Der Klassenlader stellt die

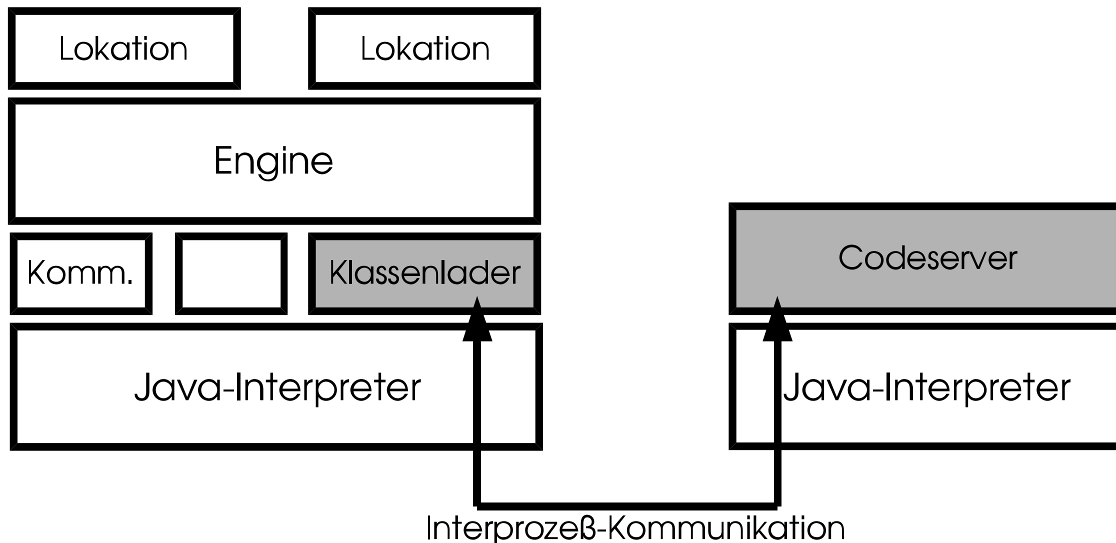

Abbildung 7: Der Codeserver als eigenständiger Prozeß, der mit dem Agentensystem über Nachrichten kommuniziert.

gewünschte Klasse der Engine zur Verfügung. Der Codeserver wird durch diese Architektur unabhängig von der Engine, die nun nach belieben gestartet und gestoppt werden kann.

Der Codeserver ist von den angeschlossenen Engines unabhängig und kann daher beliebig im Netzwerk positioniert werden:

- Um möglichst geringe Verzögerungen zu erfahren, kann der Codeserver auf dem gleichen Rechner laufen, wie die Engine. Die Kommunikation muß dabei nicht alle Schichten des Netzwerks (nach dem OSI-Modell) durchlaufen und wird dadurch deutlich schneller. Das Netzwerk wird nicht belastet und die Kapazität steht für andere Zwecke voll zur Verfügung.
- Der Codeserver kann im Netzwerk so positioniert werden, daß unnötige Replikate überflüssig werden. Jeder Codeserver speichert häufig benötigten Klassencode lokal ab. Das führt dazu, daß benachbarte Codeserver gleiche Klassen speichern, wenn diese von ihren Engines häufig verlangt werden. Dadurch kann Redundanz innerhalb eines lokalen Netzwerks entstehen. Wird nur ein Codeserver im lokalen Netzwerk eingerichtet, so wird Redundanz wirksam vermieden.
- Neben technischen Gründen kann es auch organisatorische Gründe für die Anordnung der Codeserver im Netzwerk geben.
- Vom Codeserver wird erwartet, daß er permanent am Netzwerk angeschlossen ist, für das Agentensystem trifft das nicht immer zu. Der eigenständige Server kann permanent am Netzwerk bleiben, während das Agentensystem nach belieben ans Netzwerk angeschlossen und abgehängt werden kann.

Bei den vorhergehenden Vorschlägen mußte nicht geklärt werden, wer die Codeserver administriert, da diese als Teil des Agentensystems nebenher verwaltet werden. Bei der externen Lösung ist die Zuordnung von Codeservern und Engines beliebig, es muß also sichergestellt werden, daß für jede Engine ein Codeserver günstig erreichbar ist. Es muß folglich jemand benannt werden, der für den Codeserver zuständig ist. Da ein Codeserver zwingend erforderlich ist, um das Agentensystem zu betreiben, ist der Betreiber einer Engine selbst daran interessiert einen Anbieter zu finden, der für ihn einen günstigen Codeserver zur Verfügung stellt. Steht kein günstig zu erreichender Codeserver zur Verfügung, so muß sich der Betreiber des Agentensystem auch um die Inbetriebnahme eines Codeservers kümmern. Dies darf daher keinen großen Aufwand erfordern, eventuell will der Betreiber den Codeserver auch nur für eine Übergangszeit installieren.

Da der Codeserver Ressourcen benötigt, sind damit auch Kosten verbunden, die auf die Benutzer des Servers entsprechend umgerechnet werden können. Das Problem der Ressourcenverwaltung und Abrechnungsmöglichkeiten werden in dieser Arbeit nicht weiter betrachtet, diese Aspekte finden sich in [Kuhn 1997].

Durch die externe Anbindung des Codeservers wird für jede Anfrage eine Nachricht abgesetzt. Es entsteht dadurch eine höhere Netzwerkbelastung und eine zusätzliche Verzögerung gegenüber der lokalen Anbindung. Der Klassencode wird auf einem externen Speichermedium (üblicherweise auf Festplatten) gespeichert und bei Bedarf über eine Netzwerkverbindung zum Klassenlader transportiert. Beim Einsatz eines verteilten Dateisystems ist es jedoch ohnehin der Normalfall, daß die Dateien über das Netzwerk transportiert werden.

Zeitmessungen im lokalen Netzwerk der Abteilung Verteilte Systeme ergaben die in Tabelle 3 dargestellten Zeiten. Die Zeiten unter »direkt Laden« wurden ermittelt, in dem die entsprechende Datenmenge aus einer Datei geladen wurde. Für die anderen Messung wurden zwei Programme geschrieben: einen Client und einen Server. Der Client startet die Zeitmessung und schickt dem Server eine Zahl, welche die Anzahl der erwarteten Bytes angibt. Der Server lädt aus einer Datei die gewünschte Anzahl an Bytes und schickt diese an den Client zurück, dieser stoppt nach dem Eintreffen der erwarteten Datenmenge die Zeitmessung. Die beiden Programme wurden auf verschiedenen Rechnerpaaren gestartet, die Meßergebnisse wurden durch die Wahl des Rechnerpaars nur unwesentlich beeinflußt; den Messungen lag ein verteiltes Dateisystem zugrunde.

Datenmenge	**Direkt Laden**	**Entfernt laden und Transfer per UDP**	**Entfernt laden und Transfer per TCP**
1000 Byte	3,3 ms	7,4 ms	152 ms
2000 Byte	3,3 ms	8,5 ms	182 ms
3000 Byte	3,6 ms	10,3 ms	-
4000 Byte	3,8 ms	11,4 ms	132 ms
5000 Byte	4,1 ms	12,4 ms	-
6000 Byte	4,8 ms	12,7 ms	150 ms

Datenmenge	Direkt Laden	Entfernt laden und Transfer per UDP	Entfernt laden und Transfer per TCP
7000 Byte	5,7 ms	14,5 ms	-
8000 Byte	6,4 ms	15,0 ms	166 ms
9000 Byte	6,6 ms	17,0 ms	-
10000 Byte	7,1 ms	17,2 ms	161 ms

Tabelle 3 : Gemittelte Zeitdauer für das Laden von Klassencode.

Die gemessenen Werte wurden im belasteten Netzwerk gemessen, Meßwerte von einem unbelasteten Netzwerk sind ohnehin nur von theoretischem Interesse. Es ist in der Praxis auch nicht möglich alle sonstigen Belastungen des Netzwerks auszuschalten, da die verwendeten Dienste bereits eine Grundbelastung für das Netzwerk erzeugen. Das Nachrichtenaufkommen in einem Netzwerk ist nicht konstant, so daß immer wieder ungewöhnlich hohe Meßwerte auftraten. Die Meßwerte aus Tabelle 3 sind die Mittelwerte aus 1000 Messungen. Die Werte wurden innerhalb des Java-Laufzeitsystems ermittelt, das eine Uhr mit der Auflösung von einer Millisekunde zur Verfügung stellt. Die angegebenen Meßwerte zeigen folglich eine Genauigkeit, die so nicht gemessen werden kann. Die Messung der Übertragungszeit per TCP wurde nicht für alle Datenmengen durchgeführt.

Die Meßergebnisse für das Laden einer Datei und die Übertragung mittels UDP sind im Rahmen der Meßgenauigkeit erwartungsgemäß. Das direkte Laden über das verteilte Dateisystem erfordert bereits die Übertragung der Datei über das lokale Netzwerk. Für den Transfer per UDP muß die Datei ein weiteres mal über das Netzwerk transportiert werden. Eine Verdopplung der Zeitdauer bei UDP gegenüber dem direkten Laden kann daher als untere Schranke angesehen werden. Die Zeitdauer kann verbessert werden, wenn das direkte Laden auf dem Rechner ausgeführt wird, der die gewünschten Dateien lokal verwaltet.

Die Zeitwerte für die Kommunikation über eine TCP-Verbindung scheinen dagegen keine Regelmäßigkeit aufzuweisen. Die Tabelle 3 zeigt für eine Datenmenge von 2000 Bytes den größten Zeitbedarf für größere Datenmengen scheint der Zeitbedarf kleiner zu sein. Zur Aufklärung dieses Widerspruchs wurden weitere Messungen durchgeführt, die interessantesten Meßwerte sind in Tabelle 4 zusammengefaßt. Die Tabelle zeigt für Datenmengen bis zu 1454 Byte einen Zeitbedarf von etwa 103 ms. Für größere Datenmengen dagegen lediglich einen Zeitbedarf von etwa der halben Zeit.

Datenmenge	Kommunikationsdauer
1450 Byte	104 ms
1452 Byte	103 ms
1454 Byte	102 ms
1456 Byte	105 ms
1458 Byte	52 ms

Datenmenge	Kommunikationsdauer
1460 Byte	55 ms

Tabelle 4: Einbruch der Kommunikationsdauer bei einer bestimmten Datenmenge.

Dieses Verhalten läßt sich dadurch erklären, daß der verwendete Netzwerkdienst vor dem Senden kleiner Datenmengen etwa 50 ms auf weitere Daten wartet, um so größere Datenpakete zusammenstellen zu können. Der reine Zeitbedarf für die Übertragung kann dagegen vernachlässigt werden. Der Request erreicht die benötigte Größe nicht und wird deshalb immer um ca. 50 ms verzögert. Die Antwort erreicht bei 1458 Byte Nutzdaten eine Größe, die ein weiteres Warten überflüssig macht. Das Paket wird sofort abgeschickt und die halbe Wartezeit wird eingespart, wodurch die Gesamtdauer sich halbiert. Diese Überlegung wird durch das Verhalten beim Ausliefern der Daten bestätigt: Bei Socket-Verbindungen werden in Java bei großen Datenmengen oft nur die ersten 1460 Byte ausgeliefert, der Rest kann in weiteren Leseanweisungen nachgefordert werden. Offensichtlich ist bei einer Nutzdatenmenge von etwa 1460 Byte eine Versandeinheit des darunterliegenden Netzwerks gefüllt.

Die Kommunikation mittels Verbindung kann stark beschleunigt werden, wenn der Verbindungsdienst dazu gebracht werden kann auf eine weitere Verzögerung zu verzichten. In der Regel wird für das Schreiben eines nicht vollen Puffers die Methode »`flush`« verwendet. Versuche mit dieser Methode ergab keine Änderung. Das Überprüfen der Methode, die sich in der Java-Bibliothek befindet, ergab, daß sie bislang nicht implementiert ist. Möglicherweise wird es in Zukunft eine Implementierung dieser Methode geben, so daß die Verzögerung bei der verbindungsorientierten Kommunikation nicht mehr relevant ist. Sollte diese Möglichkeit nicht in Betracht kommen, so kann der verbindungsorientierte Dienst auf Basis von UDP selbst nachgebildet werden.

3.1.5 Lokale und entfernte Anbindung an die Engine

Neben den beiden letzen Möglichkeiten ließe sich noch eine Kombination aus beiden Möglichkeiten bilden, wie sie die Abbildung 8 zeigt.

Die existierenden Codeserver sind dabei lokal an die Engine angebunden und können zusätzlich Klassenanforderungen über eine Netzwerkverbindung annehmen. Es muß jedoch nicht jede Engine über einen eigenen Codeserver verfügen.

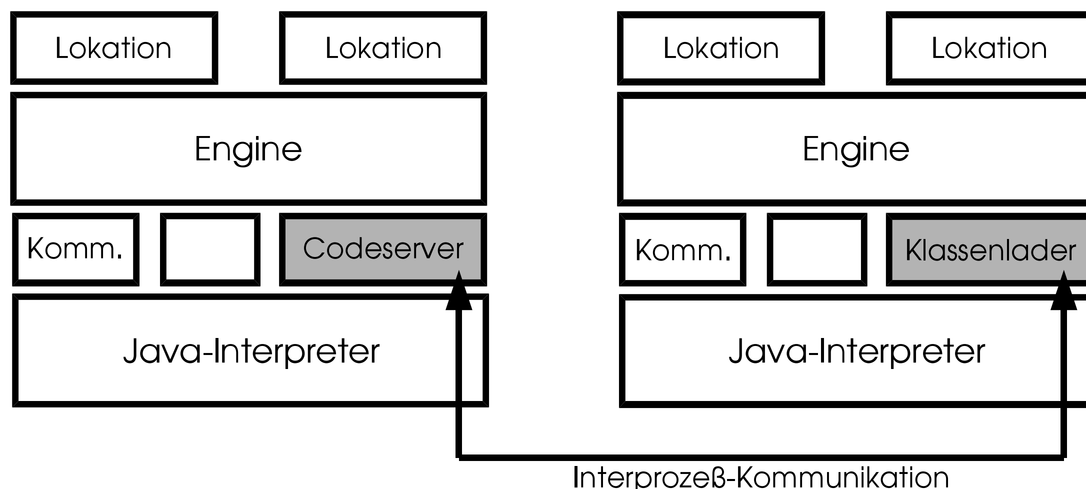

Abbildung 8: Lokale Anbindung an die Engine (links), entfernte Anbindung an die Engine (rechts).

Die Engines, die über einen eigenen Codeserver verfügen, verhalten sich wie unter 3.1.3 beschrieben. Engines ohne einen Codeserver verhalten sich dagegen wie unter 3.1.4 beschrieben, wobei statt eines eigenständigen Servers ein Server verwendet wird, der an einer anderen Engine direkt angebunden ist.

3.1.6 Vergleich der Möglichkeiten

Zusammenfassend sollen nochmals die wichtigsten Eigenschaften der verschiedenen Anbindungen aufgeführt werden. Die Anbindung des Codeservers an die Lokation wird an dieser Stelle nicht mehr berücksichtigt, da dieser Vorschlag keinerlei Vorteile gegenüber der Anbindung an die Engine besitzt.

Eigenschaft	**Anbindung an Engine**	**Kombination**	**Eigenständiger Server**
Zuordnung Server : Engine	1 : 1	1 : n	n : m (sinnvoll: n = 1)
Kommunikation Server-Engine	lokale Prozeduraufrufe	lokale und entfernte Prozeduraufrufe	entfernte Prozeduraufrufe
Probleme, wenn Engine abgebrochen wird	Codeserver wird ebenfalls abgebrochen, statistische Daten gehen dabei verloren.	Bei bestimmten Engines wird der Codeserver ebenfalls abgebrochen, statistische Daten gehen dabei verloren.	Der Codeserver läuft weiter, es muß nur eine eventuell vorhandene Verbindung geschlossen werden.

Eigenschaft	Anbindung an Engine	Kombination	Eigenständiger Server
Rechenzeitzuteilung	Der Codeserver und die Engine teilen sich eine Zeitscheibe.	Der Codeserver und die Engine teilen sich eine Zeitscheibe.	Codeserver und Engine haben eigene Zeitscheiben.
Redundanz	Innerhalb eines Rechners möglich.	Redundanz kann vermieden werden (zu Lasten der Kommunikationskosten).	Redundanz kann vermieden werden (zu Lasten der Kommunikationskosten).
Zuständigkeit	Jeder Betreiber einer Engine muß einen Codeserver betreiben.	Jeder Betreiber einer Engine kann einen Codeserver betreiben.	Die Zuständigkeit für die Codeserver ist unabhängig von der Zuständigkeit der Engines.

Tabelle 5 : Zusammenfassung der Eigenschaften der verschiedenen Möglichkeiten, den Codeserver an das Agentensystem anzubinden.

Die Tabelle 5 zeigt in der mittleren Spalte, daß die Kombination »Lokale und entfernte Anbindung an die Engine« zwar eine Reihe an Vorteilen aus den beiden anderen Vorschlägen vereint, aber dies gilt für die Nachteile leider gleichermaßen. Da dieser Vorschlag keine relevanten Vorteile bringt, ist er zu meiden, da sonst die Betreiber von Engines mit zwei verschiedenen Agentensystemen konfrontiert werden, diese Verwirrung würde bei vielen Benutzern dazu führen, daß sie unnötigerweise eine Engine mit Codeserver starten um auf der sicheren Seite zu sein.

Der größte Nachteil des eigenständigen Servers ist die Verzögerung durch die Übertragung innerhalb eines (lokalen) Netzwerkes. Es ist jedoch auch zu berücksichtigen, daß durch die gemeinsame Nutzung des Servers im lokalen Netzwerk (bei gleicher Plattenkapazität) die Wahrscheinlichkeit steigt, daß eine Klasse bereits im lokalen Netz verfügbar ist, weil ein erheblicher Teil an Redundanz beseitigt wurde und für andere Klassen zur Verfügung steht. Durch diesen Effekt kann ein Teil der Verzögerung wieder wett gemacht werden.

Eine optimale Lösung steht leider nicht zur Auswahl. Ziel sollte es sein, den Codeserver so zu implementieren, daß es mit vertretbarem Aufwand möglich ist, die Anbindung an Mole nachträglich noch zu ändern. Dies ist deutlich einfacher, wenn der Codeserver zunächst den allgemeineren Fall unterstützt und erst später spezialisiert wird. Die Lösung des eigenständigen Servers bietet die meiste Flexibilität und sollte daher als Ausgangsbasis der Implementierung dienen. Soll der Codeserver später in die Engine lokal integriert werden, so kann dann jederzeit der RPC-Mechanismus entfernt und durch lokale Prozeduraufrufe ersetzt werden.

3.2 Anbindung an Java

Die Sprache Java erlaubt zwei verschiedene Möglichkeiten, wie Klassen in das Laufzeitsystem geladen werden können.

3.2.1 Laden aus dem Dateisystem

In der vorliegenden Implementierung des Agentensystems werden die Klassen durch das Laufzeitsystem direkt aus dem Dateisystem geladen.

In Java sind alle Objekt (mit Ausnahme von Arrays) Instanzen einer Klasse. Zur Laufzeit können Objekte mit der Methode `getClass` abfragen, welcher Klasse ihre Instanz angehört. Diese Methode gibt ein Objekt der Klasse `Class`, ein Klassendeskriptor, zurück. Die Klasse `Class` besitzt eine Klassenmethode `forName`, welche den Klassendeskriptor zu einem spezifizierten Namen zurückgibt. Ist der Klassencode, der mit `forName` abgefragten Klasse, noch nicht geladen, so wird er automatisch geladen. Dazu sucht das Java-Laufzeitsystem alle Pfade im Dateisystem ab, die in der Umgebungsvariablen `CLASSPATH` angegeben sind. Wird eine Datei mit entsprechendem Namen nicht in diesen Pfaden gefunden, so schlägt die Methode mit einer `ClassNotFoundException` fehl.

Diesen Mechanismus kann der Codeserver ausnutzen um eine Klasse in das Laufzeitsystem von Java zu bringen. Nach dem Auftreten der `ClassNotFoundException` muß die Klasse, zum Beispiel über das Netzwerk, besorgt werden. Dann wird der Klassencode so im Dateisystem abgelegt, daß ihn die `forName`-Methode finden kann, das heißt die Klasse wird in einem Pfad abgelegt, der im `CLASSPATH` angegeben ist. Sobald die Klasse mit der Methode `forName` geladen wurde, kann die Datei wieder entfernt werden, denn das Laufzeitsystem greift nicht auf Dateien geladener Klassen zu. Diese Eigenschaft wurde durch praktische Versuche ermittelt, die Dokumentation von Java macht keinerlei Angaben, wann eine Klassendatei gelöscht werden kann.

Diese Art der Anbindung an Java hat einige Nachteile:

- Das Laufzeitsystem muß die Klasse finden können, der Pfad im Dateisystem wird durch den Klassennamen vorgegeben. Wenn die entsprechenden Verzeichnisse noch nicht existieren, so müssen sie erzeugt werden.
- Die Klasse muß erst gespeichert werden, um schließlich sofort wieder geladen zu werden, die unnötigen Dateizugriffe kosten Zeit.
- Da die Klassen im lokalen Dateisystem stehen, sieht das Laufzeitsystem die Klassen als vertraut an. Eine Überprüfung mit den in der Sprache Java vorhandenen Mitteln (siehe `SecurityManager` in [JavaAPI 1996]) ist nicht vorgesehen.

Wegen obiger Nachteile ist die Verwendung eines Classloaders vorzuziehen. Die Verwendung des Classloaders ist etwas komplizierter, aber auch flexibler.

3.2.2 Indirektes Laden von Klassen

Zunächst wird eine Unterklasse der abstrakten Klasse `ClassLoader` geschrieben. Die Methode `loadClass` der abstrakten Klasse muß in der eigenen Klasse implementiert werden.

Sobald ein Objekt des eigenen Classloaders angelegt ist, kann dieses Objekt verwendet werden, um Klassen in das System zu laden. Zu diesem Zweck wird die Methode `loadClass` des Objekts aufgerufen. Diese Methode kann den Klassencode von einer beliebigen Quelle besorgen, eine solche Quelle kann das Dateisystem oder auch das Netzwerk sein. Unabhängig von der Quelle wird der Klassencode in einem Array von Bytes gespeichert. Dieses Array dient als Übergabeparameter für die Systemmethode `defineClass`, welche die Klasse von seiner Superklasse `ClassLoader` geerbt hat. Die Systemmethode prüft, ob der Inhalt des übergebenen Arrays einer gültigen Codierung einer Klasse entspricht. Wenn die geladene Klasse keine formale Fehler aufweist, dann wird sie vom Java-Laufzeitsystem übernommen und steht ab sofort zur Verfügung. Die Methode `defineClass` gibt im Erfolgsfall den Klassendeskriptor der registrierten Klasse zurück; ab diesem Zeitpunkt können Objekte dieser neuen Klasse erzeugt werden.

Benötigen die erzeugten Objekte weitere Klassen, so wird durch das Java-Laufzeitsystem automatisch die Methode `loadClass` desselben Klassenladers aufgerufen, um die fehlenden Klassen zu laden. Bei diesem Vorgehen zum Laden der transitiven Klassenmenge sind höhere Schichten der Architektur, insbesondere die Engine, (siehe Kapitel 3.1 und Abbildung 4) nicht beteiligt.

4 Der Basismechanismus

Die Anforderungen (siehe »Schnittstellen«) an das Serversystem verlangen, daß das Serversystem zu einem beliebigen Klassennamen entweder den zugehörigen Klassencode besorgen oder mit Sicherheit sagen kann, daß dieser Klassenname nicht in die Codebasis aufgenommen wurde.

Die Anzahl der Lösungen für dieses Problem wird durch obige Restriktionen (siehe Kapitel »Restriktionen«) stark reduziert. Aufgrund der riesigen Menge von Klassen, die von dem Serversystem verwaltet werden sollen, ist es nicht mehr möglich alle Klassen auf einem System zu halten. Die Klassen werden also auf viele Rechner verteilt, so daß ein Mechanismus existieren muß, der den Server auffindet, auf dem die gesuchte Klasse gespeichert ist oder der bestätigen kann, daß die gesuchte Klasse nicht existiert. Es soll aber darauf verzichtet werden, eine Liste aller Klassennamen auf einem Rechner zu halten, da diese zu viel Ressourcen verbrauchen würde und nicht aktuell gehalten werden kann. Diese Bedingung verbietet für den einzelnen Server den Aufbau eines Verzeichnisses, das zu jeder Klasse eine oder eine Menge von Rechneradressen verwaltet.

Wenn die Klassen mit beliebigen Namen auf dem Server verbleiben sollen, auf dem sie registriert wurden, so kann eine beliebige Klassenanfrage nur durch Absuchen aller Server mittels Multicast beantwortet werden. Um Multicasts zu vermeiden läßt sich mittels eines Tricks der Klassenname so ändern, daß der Heimatserver direkt gefunden werden kann. Eine weitere Lösung bindet die Codemigration an die Migrationspfade der Agentenobjekte. Zwei Möglichkeiten beschäftigen sich mit einer verteilten Lösung der Namensauflösung. Die letzte hier vorgestellte Lösung sieht eine völlig verteilte Verarbeitung der Klassen vor, die Server müssen dafür eng miteinander kooperieren. Abschließend werden die Lösungsvorschläge verglichen und ein geeigneter Mechanismus zur Implementierung ausgewählt.

4.1 Bestätigtes Propagieren

Dieser Ansatz ist eine Erweiterung des bereits bestehenden Mechanismus. Die prinzipielle Idee, die hinter diesem Ansatz steckt, sieht für den Klassencode dieselben Migrationspfade wie für die Agentendaten vor.

Beim bestätigten Propagieren wird davon ausgegangen, daß die Anforderungen an bestimmten Klassencode nicht zufällig auftritt. Vielmehr bedingt dieser Ansatz, daß eine Klasse K (zum Beispiel ein bestimmter Agent) nur Klassen aus einer beliebigen aber festen Menge Ref(K) referenziert. Der Ansatz verlangt die Erhaltung folgende Invarianten:

1. Wenn K lokal vorhanden ist, dann müssen auch alle Klassen, die in Ref(K) enthalten sind, lokal vorhanden sein.
2. Solange sich ein Agent der Klasse AK an einem Ort befindet, muß die Klasse AK am zugehörigen Server lokal gehalten werden.
3. Ein Agent der Klasse AK verläßt einen Ort erst, wenn die Klasse AK beim Server des Zielorts lokal vorgehalten wird.

Aus der ersten Invariante folgt, daß beim Eintreffen eines Agenten alle Klassen, die der Agent benötigt, angefordert werden müssen. Alle diese Klassen sind aufgrund der zweiten Invariante am letzten Aufenthaltsort des Agenten vorhanden. Erst wenn der neue Ort bestätigen kann, daß alle benötigten Klassen vorhanden sind, kann der Agent den Ort verlassen. Der Server kann die zugehörige Klasse erst dann löschen, wenn der letzte Agent der Klasse sich beendet oder bestätigt wurde, daß der Agent den Ort verlassen hat.

Obwohl der Ansatz einfach und logisch erscheint, so kann die Voraussetzung nicht immer sinnvoll erfüllt werden. So wird doch vorausgesetzt, daß die Menge der referenzierten Klassen beliebig, aber fest ist. Wie bereits in Kapitel »Eigenschaften des Codes mobiler Agenten « festgestellt wurde, ist die maximale Klassenmenge zu einem festen Zeitpunkt nicht entscheidbar. Die Menge Ref(K) muß eine endliche Obermenge der maximalen Klassenmenge sein, sie ist daher ebenfalls nicht entscheidbar. Das bestätigte Propagieren stellt also nur für die festgelegten referenzierten Klassen eine Lösung dar, da diese Einschränkung die Mächtigkeit der Programmiersprache Java zu stark einschränken würde, ist das bestätigte Propagieren kein geeigneter Basismechanismus. Das Propagieren von Klassen entlang der Migrationspfade der Agenten kann dennoch angewandt werden, wenn dadurch eine Verkürzung der Migrationszeit zu erwarten ist, im Kapitel »Verbesserung der Performanz« wird dieser Ansatz deshalb näher beschrieben.

4.2 Multicast

Wenn kein Zusammenhang zwischen dem Klassennamen und der Serveradresse bestehen soll, dann kann eine Klasse mit beliebigen Namen auf einem beliebigen Server gehalten werden. Bei der Suche nach einer Klasse kann der Suchraum nicht aufgrund des Namens eingeschränkt werden, die Klasse könnte auf jedem noch nicht abgesuchten Server liegen. Ein Suchverfahren kann nur garantieren, daß eine nicht gefundene Klasse nicht auf den durchsuchten Servern liegt. Das bedeutet, daß im schlechtesten Fall alle Server betrachtet werden müssen. Insbesondere tritt dieser Fall ein, wenn nach einer Klasse gesucht wird, die nicht registriert wurde.

Die Realisierung eines solchen Suchverfahrens kann durch einen Broadcast vorgenommen werden. Bei einem großen System läßt sich schnell errechnen, daß viele Broadcast pro Sekunde zu erwarten sind, diese Flut von Anfragen und Nachrichten beim Broadcast ist nicht mehr zu bewältigen.

Deshalb wird die Suchnachricht nicht an alle Rechner gesendet, sondern nur an die Server. Das Problem könnte auch durch einen Eventmanager gelöst werden [Beck 1997]. Alle Server melden sich beim Eventmanager für das Ereignis an, das eintritt, wenn eine Engine eine Klasse benötigt. Die Weiterleitung eines Events geschieht effizient entlang eines (minimal) spannenden Baumes, dieser besitzt die Eigenschaft, daß er alle Server miteinander verbindet und die Summe seiner Kantengewichte, hier die Verzögerung der Netzwerkverbindungen, minimal ist. Trifft die Anfrage bei einem Server ein, der die Anfrage beantworten kann, so bricht dieser Server den Multicast in seinen Unterbäumen ab; die Anfrage kann dennoch in anderen Teilbäumen bis zum Ende laufen.

Wenn man annimmt, daß die Suche meistens durch einen wenig entfernten Server beantwortet werden kann, dann ist es möglich den Aufwand an Nachrichten zu verringern. Zum Beispiel ließe sich der Multicast in seiner Reichweite begrenzen, indem jede Anfrage zunächst nur eine begrenzte Zahl von Rechnern erreicht. Wenn sich auf diese Anfrage noch kein Server meldet, so kann die Suche mit einer erweiterten Reichweite wiederholt werden.

Die Zahl der Nachrichten zum Suchen einer Klasse bei n Server beträgt im schlechtesten Fall exakt n - 1, da alle anderen Server eine Nachricht erhalten müssen. Der mittlere Fall kann nicht abgeschätzt werden, da die Ausprägung des Baums einen entscheidenden Einfluß hat. Dazu zwei Beispiele:

1. Der Baum mit n Servern besteht aus einer Wurzel mit dem Grad n - 1, das heißt alle anderen Server hängen direkt an der Wurzel. Eine nicht-lokale Anfrage kann in der Wurzel starten, dann ist die Zahl der Nachrichten exakt n - 1. Startet die Anfrage bei einem Blatt, dann kann das Ziel die Wurzel oder ein anderes Blatt sein, im ersten Fall wird nur eine Nachricht, im zweiten Fall n - 1 Nachricht benötigt. Für eine große Zahl von Servern strebt die mittlere Anzahl benötigter Nachrichten nach n - 1.

2. Der Baum ist linear, das heißt jeder Server kennt seine beiden Nachbarn (mit Ausnahme der beiden Rändern). Die Suchnachricht wird vom startenden Server in beiden Richtungen verbreitet, bis ein Server auf die Nachricht antworten kann. Unter der Annahme, daß genau ein Server auf die Anfrage antworten kann, kann die mittlere Zahl von Nachrichten wie folgt ermittelt werden: Die Serverkette I kann in zwei Hälften geteilt werden, die Anfrage kann entweder auf der gleichen Hälfte beantwortet werden oder erst in der anderen Hälfte. Im letzteren Fall müssen alle Server der erste Hälfte benachrichtigt werden. Im ersten Fall kann die Hälfte wieder halbiert werden und die Betrachtung beginnt erneut. Durch Auflösen der Rekursionsgleichung $N(I)=\frac{1}{2}\left(\frac{1}{2}+\frac{1}{4}I\right)+\left(\frac{1}{4}I+N\left(\frac{I}{2}\right)\right)$ erhält man $\frac{2}{3}I$ Nachrichten, es werden also im Mittel zwei Drittel aller Server im betrachteten Intervall an der Suche beteiligt.

Die Verwaltung des spannenden Baums erlaubt problemlos das An- und Abmelden von Servern. Bei Ausfall und Netzpartinionierung muß der spannende Baum allerdings neu konfiguriert werden, zumindest ist dies möglich. Die Anordnung in einem Baum ist dagegen weniger günstig, da im Baum praktisch jeder Knoten einen Flaschenhals darstellt.

Klassen können beliebig auf den Servern repliziert werden. Mit steigender Zahl an replizierten Klassen nimmt die Zahl der benötigten Nachrichten ab, dennoch ist die Zahl der Nachrichten zu groß.

4.3 Beheimatete Klassen

Gegenüber den Voraussetzungen, die zu der Idee des Multicasts führten, wird bei den beheimateten Klassen eine Verbindung zwischen dem Namen und dem Ort des Servers, auf dem die Klasse zu finden ist, hergestellt.

4.3.1 Der Mechanismus

In diesem Ansatz wird der Name der Klasse erweitert. Der Server bei dem die Klasse angemeldet wird, ist die *Heimat* der Klasse. Dem Klassennamen wird eine geeignete Codierung der Adresse des Heimatservers als Präfix vorangestellt. Die Namenskonventionen von Java ([JavaSpec 1997] siehe Kapitel »7.7 Unique Package Names«) müssen dabei eingehalten werden. Nach der Anmeldung ist die Klasse nur noch unter dem neuen (erweiterten) Namen erreichbar. Wird nach einer Klasse verlangt, dann wird der Präfix des Klassennamens abgelöst und daraus die Adresse des Heimatservers zurückgewonnen. Der mit dieser Adresse

erreichte Server ist die Heimat der Klasse, so daß auf diesem Server die Klasse garantiert vorhanden sein muß.

Für jede Klasse ist bereits aufgrund des Klassennamens eindeutig, wo diese Klasse zu finden ist. Mit einem einfachen Zugriff kann die Klasse aus Ihrer Heimat besorgt werden, oder bestätigt werden, daß die gesuchte Klasse nicht existiert. Die Zahl der Nachrichten zum Auffinden einer beliebigen Klasse ist folglich konstant, in der Regel ist nur eine Nachricht nötig.

Neben dem Auffinden des Heimatservers, wird durch die beheimateten Klassen gleich ein weiteres Problem gelöst: Es gibt keine globalen Konflikte von gleichen Klassennamen. Beim Registrieren von neuen Klassen muß zunächst geprüft werden, ob die Klassen nicht bereits im Serversystem vorhanden ist. Diese Prüfung kann bei der Heimat der Klasse lokal durchgeführt werden, da nur Klassen dieses Servers Namenskonflikte hervorrufen können.

Neue Server können jederzeit in Betrieb genommen werden, da dies keinen Einfluß auf die anderen Server hat. Der neue Server ist nur für die bei ihm beheimateten Klassen zuständig.

Das Abmelden eines Servers ist dagegen problematisch, da dann die auf ihm beheimateten Klassen nicht mehr zur Verfügung stehen. Das ist insoweit akzeptabel, da von einem Server vorausgesetzt wird, daß er ständig im Netzwerk präsent ist. Dennoch ist es immer wieder notwendig, auch Server zeitweise zu beenden. Es sollte nun eine Möglichkeit geben, die Klasse bei zeitweiliger Nichtverfügbarkeit des Heimatservers zu finden und zu besorgen. Beim Abmelden eines Servers kann dieser noch Vorkehrungen treffen, bei Ausfall eines Servers und bei Netzpartitionierung ist dies nicht mehr möglich, die entstehende Problematik ist prinzipiell die gleiche.

4.3.2 Replikate

Um die Verfügbarkeit einer Klasse nicht von der Präsenz eines einzigen Servers abhängig zu machen, werden die Klassen auf mehreren Servern abgelegt. Jede Klasse besitzt dennoch genau eine Heimat, die anderen Server sind nur *Sekundärserver*. Die Vergabe der Klassennamen wird durch die Sekundärserver komplizierter, der Name, wie er bislang vergeben wurde, wird als *echter Name* bezeichnet. Der echte Name der Klasse ist global eindeutig und nicht redundant, das heißt daß dieselbe Klasse nicht zwei verschiedene echte Namen bekommen kann, umgekehrt gibt es zu jedem echten Namen genau eine Klasse.

Ein Beispiel soll die Begriffe verdeutlichen. Die Klasse »BeispielPackage.BeispielKlasse«[4] wurde auf dem Server mit der IP-Adresse 11.22.33.44 und der Portnummer 1234 registriert. Eine Kopie der Klasse wird auf dem Rechner mit der IP-Adresse 55.66.77.88 und Portnummer 5678 gehalten.

4 Im Quelltext werden die Teile im Namen durch Punkte abgesetzt. Das Class File Format sieht jedoch das »/« als Trennzeichen vor. Da der Server ausschließlich mit dem Class File Format arbeitet, wird dessen Syntax verwendet.

Art des Namen	Beispiel
ursprünglicher Name, Originalname	`BeispielPackage/BeispielKlasse`
echter Namen	`IP11d22d33d44/p1234/BeispielPackage/BeispielKlasse`
globaler Name	`IP55d66d77d88/p5678/IP11d22d33d44/p1234/BeispielPackage/BeispielKlasse`

Tabelle 6 : Verdeutlichung der Begriffe Originalname, echter Name und globaler Name an einem Beispiel

Der *globale Name* einer Klasse ist der echte Name der Klasse erweitert um einen Präfix. Der Präfix besteht aus einer Anreihung von geeignet kodierten Adressen, diese Adressen müssen Adressen von Sekundärservern sein. Der Präfix muß aber nicht aus den Adressen aller Sekundärserver bestehen. In der erweiterten Backus-Naur-Form ließe sich die Klassennamen wie folgt darstellen:

globalerName = {SekundärAdresse »/«} echterName.

echterName = HeimatAdresse »/« OriginalName.

HeimatAdresse = SekundärAdresse.

SekundärAdresse = »IP« Byte »d« Byte »d« Byte »d« Byte »/p« Port.

Der OriginalName wird durch den Benutzer (Programmierer) gemäß den Java Namenskonventionen vergeben, mit der Einschränkung, daß keine Klasse und kein Package mit den Buchstaben »IP« beginnen darf.

Die HeimatAdresse und die SekundärAdresse sind gleichgesetzt, da sie sich nicht syntaktisch sondern nur von der Bedeutung unterscheiden. Die Definition von »Byte« und »Port« wurde hier nicht aufgeführt es handelt sich dabei um die dezimale Zahlendarstellung im Bereich 0 bis 255 beziehungsweise im Bereich 0 bis 65536. Die Trennzeichen »IP«, »d« und »p« sind willkürlich gewählt, es können alle Zeichen zur Trennung verwendet werden, die den Java Namenskonventionen entsprechen und eine einfache lexikalische Analyse des Namens erlauben.

Treffen bei einem Server Anfragen ein, deren globaler Name noch unbekannte Sekundärserver enthält, so kann er diese in den globalen Namen seines Replikats der Klasse aufnehmen. Kennt der Heimatserver die Klasse aus obigem Beispiel nur unter dem echten Namen, dann kann der Heimatserver den Namen in den obigen globalen Namen ändern, wenn eine Anfrage diesen Namen verwendet hat. Auf diese Weise kann der Heimatserver die Kenntnis des Sekundärservers speichern und verbreiten. Auch die in anderen Klassen enthaltenen Referenzen auf die Beispielklasse können um die Adresse des Sekundärservers erweitert werden.

Wird eine Klasse aufgrund einer Anfrage ausgeliefert, so muß sie natürlich genau dem Namen entsprechen, wie er in der Anfrage verwendet wurde, das heißt, daß die Liste der

Sekundärserver eventuell gekürzt werden muß. Entspricht der Klassenname nicht dem Name in der Anfrage, dann wird die Klasse vom Java-Laufzeitsystem nicht akzeptiert. Die Optimierung der Verbreitung von Sekundäradressen ist folglich durch das Verhalten des Java-Laufzeitsystems stark eingeschränkt. Das Ziel des Basismechanismus ist jedoch nicht die Optimierung von Zugriffen, sondern die Sicherstellung, daß zu einem gegebenen Klassennamen eine entsprechende Klasse gefunden wird, wenn diese existiert.

Die Informationen über Sekundärserver können auch separat verwaltet werden, diese Möglichkeit wird im Kapitel »Beheimatete Klassen und DNS« erläutert werden.

4.4 Verwenden des »Domain Name Service«

Bei dem Domain Name Service (DNS) handelt es sich um eine verteilte Datenbank zur Abbildung von Namen auf Adressen des Internets. Der DNS kann auf zwei Arten verwendet werden, als Erweiterung der beheimateten Klassen oder zur Auflösung der Klassennamen.

4.4.1 Komplette Namensauflösung durch den DNS

Die einfachste Lösung für die Namensauflösung sieht vor, die Klassennamen direkt in den DNS einzutragen. Zu einem Klassennamen kann beim DNS die Adresse des Servers und der Sekundärserver erfragt werden.

So einfach diese Lösung auch erscheint, so ungeeignet ist die Lösung wegen der organisatorischen und der technischen Schwierigkeiten.

Das Eintragen eines Rechners in den DNS ist nicht jedem Benutzer möglich. Die Eintragung in den DNS ist ein Prozeß für den eine Automatisierung nicht vorgesehen ist. Während die organisatorischen Schwierigkeiten gelöst werden können, so gilt dies nicht für die technischen Schwierigkeiten. Das Agentensystem erfordert eine möglichst schnelle Verfügbarkeit der Klassen, wenn diese registriert wurden. Dafür müssen die Informationen möglichst schnell ausgetauscht werden, die Änderungen im DNS propagieren nicht schnell genug für eine sofortige Verfügbarkeit der Klassen nach der Registrierung.

Der DNS ist hierarchisch organisiert und kann nur Namen auflösen, die entsprechend diesen Organisationseinheiten gebildet wurden (siehe [Terry 1985]). Werden die Klassennamen

den Namenskonventionen entsprechend angepaßt, so erhält man eine Kombination aus dem Basismechanismus der beheimateten Klassen und der Verwendung des DNS.

4.4.2 Beheimatete Klassen und DNS

Die beheimateten Klassen erhalten neue Namen, deren Präfix die Adresse des Heimatservers bildet. Anstatt der festen Bindung der Klassennamen an eine bestimmte Internetadresse ist es möglich einen zusätzlichen Abbildungsschritt einzubauen. Der Name der Klasse erhält dabei als Präfix den Rechnername, wie er im DNS eingetragen ist. Bei einer Suchanfrage wird aus dem Klassennamen der Rechnername gewonnen, welche mit dem DNS auf eine Internetadresse abgebildet wird.

Gegenüber den normalen beheimateten Klassen wird durch diese zusätzliche Indirektionsstufe mehr Flexibilität erreicht. Soll ein Server verlegt werden und trägt eine andere Internetadresse, so genügt eine Änderung im DNS um den Rechner an der neuen Position ausfindig zu machen.

Allerdings wird der Name der Klasse durch den DNS Namen deutlich länger. Wenn zu einer Klasse eine Reihe von Sekundärservern bekannt sind, kann der Klassenname leicht eine unübersichtliche Länge erreichen. Während die IP Adresse, wie sie bei den beheimateten Klassen verwendet wurde, eine maximale Länge von 2+4*3+3 = 17 Zeichen hat, so ist die Länge des DNS Namens, zum Beispiel »cembalo.informatik.uni-stuttgart.de«, deutlich länger.

Damit die Klassennamen besser lesbar sind, wird hier die Trennung der Zusatzinformationen vom Klassennamen vorgeschlagen. Die nachfolgende Syntax wird durch ein Beispiel in Tabelle 7 verdeutlicht.

echterName = HeimatDNSName »/« OriginalName.

HeimatDNSName = domain {»/d« domain} »/p« Port.

Der OriginalName wird vom Benutzer vergeben. Die Trennzeichen »/d« und »/p« erlauben eine einfache lexikalische Analyse des Namens: wird im HeimatDNSName die Endung beginnend mit »/p« abgeschnitten und alle »/d« durch Punkte ersetzt, so erhält man den Namen, wie er im DNS eingetragen ist.

Art des Namen	Beispiel
ursprünglicher Name, Originalname	`BeispielPackage/BeispielKlasse`
echter Namen	`cembalo/dinformatik/duni-stuttgart/dde/p1234/BeispielPackage/BeispielKlasse`

Art des Namen	Beispiel
globaler Name	wie echter Name

Tabelle 7: Verdeutlichung der Begriffe Originalname, echter Name und globaler Name an einem Beispiel

Die Information über vorhandene Sekundärserver ist nicht mehr im Namen der Klasse vorhanden und muß deshalb separat verwaltet werden. Dazu heftet der Codeserver an jede Klasse ein Objekt, das die zusätzlichen Informationen speichert. Bei der Weitergabe der Klasse müssen auch die Zusatzinformationen der Klasse weitergegeben werden.

Die Zusatzinformation enthält zu einer beliebigen Anzahl Referenzen eine beliebige Anzahl an Sekundärservern. Zunächst werden nur festgelegt referenzierte Klassen, für die ein Sekundärserver vorhanden ist, in den Zusatzinformation gespeichert. Mit dem Bekanntwerden neuer Sekundärserver oder neuer Laufzeitreferenzen der Klasse kann die Zahl der Referenzen wachsen. Die Zahl der bekannten Sekundärserver kann selbstverständlich auch wachsen. Die zeigt, daß die Zusatzinformationen in zwei Richtungen (nach rechts und nach unten) beliebig erweitert werden können.

Einige grundlegenden Klassen werden häufig eingesetzt werden. Das bedeutet, daß diese Klassen häufig in anderen Klassen referenziert werden. Zu jeder dieser Referenzen müßte eine Liste mit den Sekundärservern gehalten werden, dies führt jedoch zu einer Speicherverwendung. Die Aktualisierung der Listen mit Sekundärservern stellt ein weiteren Nachteil dar, müßte doch nach dem Bekanntwerden eines neuen Sekundärservers einer Klasse alle Referenzen auf diese Klasse gesucht werden, damit deren Listen um den Sekundärserver erweitert werden können.

Die Verwaltung der Sekundärserver einer Klasse erfolgt im Codeserver zentral. Die Zusatzinformation hält dann nur noch eine Liste mit Verweisen auf die zentral gehaltenen Listen mit Sekundärservern. Die Liste von Sekundärservern existiert im gesamten Codeserver nur einmal, Ergänzungen werden zentral durchgeführt und sind sofort für alle Referenzen auf die Klasse gültig.

Das Trennen der Infromationen über Sekundärserver vom Klassennamen hat einige Vorteile:

- Nach der Registrierung müssen die Klassen nicht mehr verändert werden.
- Die Namen der Klassen werden kürzer und für den Benutzer übersichtlicher.
- Der Klassenname ist eindeutig und ändert sich nicht mehr.

Allerdings ist die Angabe von Sekundärservern bei Laufzeitreferenzen zunächst nicht möglich. Wird eine neue Klasse das erste mal referenziert, dann muß sich der Codeserver an den Heimatserver wenden, da ihm noch keine Informationen über Sekundärserver vorliegen, da die Angabe von Sekundärservern im Klassennamen nicht mehr möglich ist. Wurde eine Klasse referenziert, so legt der Codeserver einen Eintrag in der Zusatzinformation der Klasse an und kann dort die Adressen der Sekundärserver speichert, die ihm der Heimatser-

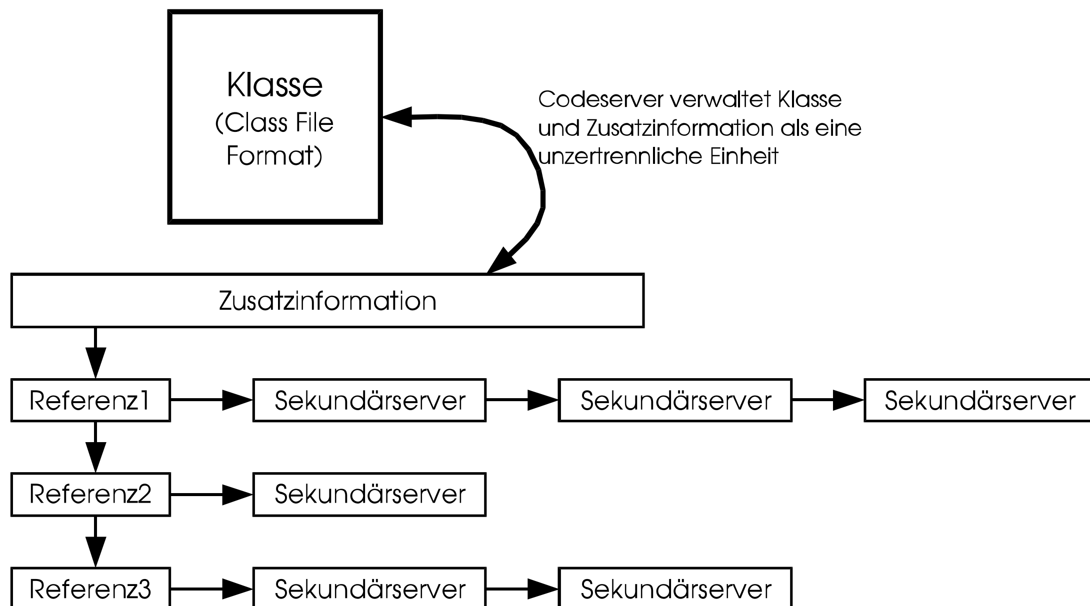

Abbildung 9: Klasse und ihre Zusatzinformation

ver mitteilt. Bei der ersten Verwendung einer Klasse, die erst zur Laufzeit referenziert wird, besteht also keine Möglichkeit auf Replikate zuzugreifen.

4.5 Verteilter Multibaum

Der verteilte Multibaum entstand aus der Idee, eine binäre Suche über alle Server durchführen zu wollen. Der verteilte Multibaum entspricht einem Binärbaum, dessen Knoten die Server darstellen. Um den Engpaß an der Wurzel zu vermeiden, ist jeder Server auf jeder Ebene vorhanden. Dadurch dient jeder Knoten gleichzeitig als Wurzel und als Blatt. Die Operationen zum Einfügen von neuen Servern und Löschen von bestehenden Servern soll die Struktur möglichst ausgeglichen halten, so daß die Zahl der Nachrichten beim Suchen von Klassen möglichst gering bleibt. Zunächst wird die Grundstruktur des verteilten Multibaums betrachtet, anschließend werden die Zugriffs- und Änderungsoperationen beschrieben.

4.5.1 Die Grundstruktur des verteilten Multibaums

Es seien $n = 2^m$ Server gegeben. Diese Server werden durchnumeriert von 0 bis n - 1. Die logische Anordnung[5] der Server soll einen geschlossenen Ring bilden, so daß auf den Server mit der Nummer n - 1 unmittelbar der Server mit der Nummer 0 folgt. Korrespondierend mit dieser Anordnung, wird bei Rechnungen mit der Nummer des Servers immer modulo n (also im modulo Ring) gerechnet.

Jeder Server kennt nur die Adressen der Server mit den Abständen n/2, n/4, n/8, ..., 1. Damit kennt jeder Server also nur 2m - 1 andere Server.

Die Klassen werden so auf die Server verteilt, daß jeder Server eine Menge von Klassen hält, die alphabetisch aufeinanderfolgen. Damit besitzt jeder Server ein Intervall aus dem Namensraum der Klassennamen, für das er zuständig ist. Die Intervalle stoßen lückenlos aneinander. Der Server mit der nachfolgenden Servernummer ist auch für das nachfolgende Intervall zuständig. Zum Beispiel könnte der Server mit der Nummer 0 die Klassen mit den Namen halten muß, die in alphabetischer Ordnung am kleinsten sind. Mit wachsender Nummer des Server werden auch die Klassennamen (in alphabetischer Ordnung) größer. Die Datenmenge, die jeder Server speichert, wird also ungefähr gleich groß angenommen.

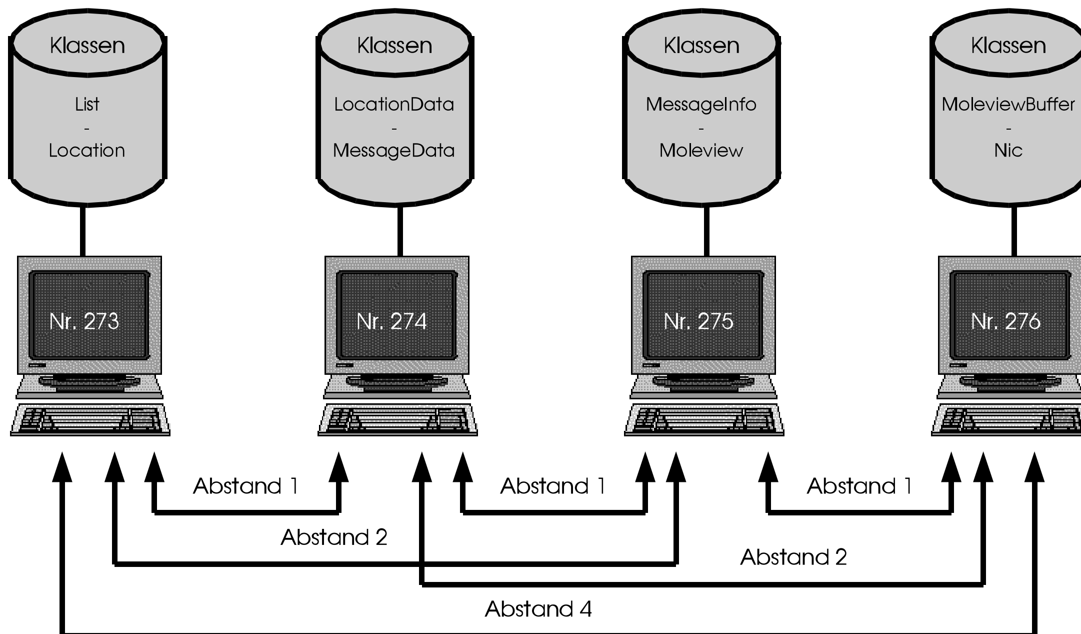

Abbildung 10: Ausschnitt aus dem verteilten Multibaum.

Die Abbildung 10 zeigt einen Ausschnitt aus einem verteilten Multibaum. Dargestellt sind die vier Server mit den Nummern 273 bis 276. Jeder der Server hat direkten Zugriff auf einige Klassen, die Teil der Codebasis des Serversystems sind. Die Klassen sind aufsteigend

5 Die logische Anordnung bezieht sich nur auf die Datenstruktur des verteilten Multibaums. Diese Anordnung ist folglich unabhängig vom geographischen Standort des Rechners oder der Anordnung im Netzwerk.

alphabetisch sortiert und es gibt keine Klassen, die zwischen den beiden Intervallgrenzen benachbarter Server liegen. Die Pfeile unterhalb der Server stellen dar, welche Adressen der Server kennt. Pfeile von oder zu Servern, die nicht abgebildet sind, wurden weggelassen.

Die beschriebene Grundstruktur setzt voraus, daß alle n Nummern an Server vergeben sind. Diese Annahme ist nicht realistisch, deshalb wird in den folgenden Abschnitten noch gezeigt, wie sich die Struktur verhält, wenn sie nicht vollständig ist.

4.5.2 Suchen im verteilten Multibaum

Der Sinn der komplizierten Grundstruktur wird sich nun zeigen, denn die Hauptaufgabe ist die effiziente Suche nach Klassen in einem großen Netzwerk. Eine Anfrage nach einer Klasse kann bei einem beliebigen Server starten.

Der Server x sei der Server, bei dem eine Anfrage nach einer Klasse initiiert wurde. Der Server untersucht zunächst ob die gewünschte Klasse in seiner Datenbasis enthalten sein müßte, dies ist der Fall, wenn der Name der gesuchten Klassen innerhalb des Intervalls liegt, das der Server verwaltet. Wenn dieser Fall zutrifft, dann kann die Anfrage sofort (positiv oder negativ) beantwortet werden.

Interessanter ist der Fall, daß ein beliebiger anderer Server für diese Klasse zuständig ist, dann wird die Nummer des zuständigen Servers mittels Intervallschachtelung ermittelt. Der Server x kann noch kein Intervall angeben, in dem sich die gesuchte Klasse befindet. Der Server x schickt die Anfrage weiter an den Server mit dem Abstand n/2, also an den Server x + n/2 (=: y).

Der Server y prüft zunächst, ob er die gesuchte Klasse verwaltet. Ist dies nicht der Fall, dann initialisiert Server y das Serverintervall mit [x + 1, y - 1], wenn der Klassenname unterhalb seines Intervalls liegt, anderenfalls mit [y + 1, x - 1]. Wird die gesuchte Klasse im ersten Intervall vermutet, dann wird die Nachricht und das Intervall an den Server mit dem Abstand n/4, also an Server y - n/4 weitergegeben. Anderenfalls geht die Nachricht an den Server mit der Nummer y + n/4. Dieser Server (z) liegt genau in der Mitte des verbleibenden Intervalls.

Der Server z prüft zuerst ob er für die Klasse zuständig ist, wenn nicht, dann kann er das Intervall halbieren, indem er die Ober- oder Untergrenze des Intervalls auf seine Nummer (∓ 1) setzt. Dann addiert er $\pm$ n/8 (das ist die halbe Intervallänge) zu seiner Nummer und schickt die Anfrage samt Intervall zu diesem Server.

Das Verfahren endet, wenn ein Server die Anfrage erhält, der für die Verwaltung der gesuchten Klasse zuständig ist. Da die Klassen lückenlos aneinander stoßen muß dieser Server zwangsläufig in maximal m Schritten gefunden werden. Durch einige Tricks kann die Zahl der Schritte und damit die Anzahl an Nachrichten, die zum Finden einer Klasse nötig sind, weiter verringert werden.

Die Schrittzahl kann deutlich verkleinert werden, wenn jeder Server nicht nur die Adressen der anderen Server kennt, sondern sogar das Intervall kennt, für das der jeweilige Server zuständig ist. Der Server kann aufgrund dieser Information die Lücke zwischen den ihm bekannten Servern finden, in der die Klasse zu finden sein müßte. Das Intervall wird durch jede Nachricht mindestens auf ein Viertel verkürzt, da der größte Abstand zweier bekannter Server genau ein Viertel der bisherigen Intervallänge beträgt. Das heißt im schlechtesten Fall kann die Schrittzahl dadurch halbiert werden auf ½ $\log_2 n$. Im durchschnittlichen Fall beträgt die Schrittzahl sogar nur noch ein Drittel, also 1/3 $\log_2 n$ (ohne Nachweis). Ein Beispiel verdeutlicht die Schrittzahl eindrucksvoll: Für ein Serversystem mit 32700 Servern werden maximal 8 und durchschnittlich nur 5 Nachrichten zum Auffinden des zuständigen Servers benötigt.

4.5.3 Aufnehmen neuer Klassen

Die Aufnahme neuer Klassen funktioniert zunächst genau gleich, wie eine Anfrage. Ist der zuständige Server lokalisiert, dann kann die Klasse dort einfach installiert werden. Liegt der Klassenname zwischen zwei Klassennamen verschiedener Server, dann nimmt der Server mit der geringeren Auslastung die Klasse auf.

Wird bei der Aufnahme einer Klasse die Kapazität eines Servers überschritten, so gibt er die Klassen an seine beiden direkten Nachbarn ab, die an den Rändern seines und deren Zuständigkeitsbereich liegen. Solange das Gesamtsystem genügend Kapazität aufweist, ist es mit dem beschriebenen Verfahren möglich, die Klassen zu verteilen. Ein Server könnte sogar Klassen abgeben, wenn er zwar genug Kapazität hätte, jedoch nicht schnell genug alle Anfragen bearbeiten kann.

Eine Klasse kann analog aus dem System gelöscht werden, jedoch tritt dann nicht der Fall auf, daß die Kapazität eines Servers überschritten wird. Eine schwache Auslastung eines Servers ist sicherlich erträglich, solange andere Server nicht überlastet sind. Dieses Vorgehen bei der Verteilung der Klassen entspricht den Sender-initiierten Verfahren bei der Lastbalancierung in Verteilten Systemen.

4.5.4 Vergabereihenfolge der Servernummern

Das System scheint zunächst nur zu funktionieren, wenn die Anzahl der Server eine Zweierpotenz darstellt. Diese Bedingung kann jedoch nicht eingehalten werden, da es jederzeit möglich sein muß neue Server in das System zu integrieren und andererseits Server aus dem System herausgenommen werden sollen. Das Problem kann nicht trivial gelöst werden, da garantiert werden muß, daß keine Lücke in der logischen Anordnung der Server entstehen. Das Zulassen von unbesetzten logischen Nummern im verteilten Multibaum zerstört das günstige Zugriffsverhalten und läßt den Baum entarten.

Das nachfolgende Verfahren verhindert die Bildung von Lücken und läßt eine beliebige Skalierung des Serversystems zu, soll aber zur Vereinfachung hier auf 2^{32} Server beschränkt werden. Die gewählte Beschränkung ist so dimensioniert, daß sie kurz- und mittelfristig nicht zu Problemen führt wird.

Alle beteiligten Server besitzen jeweils eine eindeutige Nummer zwischen 0 und 2^{32} - 1. Die Nummern werden nicht aufsteigend vergeben sondern gemäß folgender Reihenfolge.

1. Zuerst wird die Null vergeben.
2. Es werden alle Nummern vergeben (in diesem Fall nur eine), die Vielfache von 2^{31} sind.
3. Es werden alle Nummern vergeben, die Vielfache sind von 2^{30} sind.
4. Es werden alle Nummern vergeben, die Vielfache von 2^{29} sind.
5. Und so fort, bis alle Vielfachen von 1 vergeben werden.

Jede Nummer wird selbstverständlich nur einmal vergeben, da alle Vielfachen von 2^x auch Vielfache von 2^{x-1} sind, wird nur jede ungerade Vielfache vergeben.

Die Vergabe der Vielfachen einer Schicht kann nicht einfach aufsteigend vorgenommen werden, da sonst der Multibaum nicht immer ausgeglichen bleibt. In jeder Schicht werden genau so viele Nummern vergeben, wie in allen darüberliegenden Schichten. Würden die Nummern einer Schicht in aufsteigender Reihenfolge vergeben, dann wäre nach der Vergabe der Hälfte der Nummern einer Schicht in dem Intervall von Null bis aus 2^{31} doppelt so viele Nummern vergeben, wie im restlichen Intervall. Die Vergabe muß folglich symmetrisch zur Mitte des Intervalls erfolgen, damit die Zahl der vergebenen Nummern auch während der Vergabe in allen vergleichbaren Intervalle möglichst gleich groß ist. Es gibt Funktionen, die die Vergabe der Nummern ausgeglichen vornehmen, ihre Darstellung ist an dieser Stelle zu aufwendig, da die Realisierung der Funktion für die Funktion des verteilten Multibaums nicht von Bedeutung ist.

Jeder Server besitzt Platz für 63 Adressen anderer Server. Je zwei Adressen werden benötigt für Server mit den Abständen 1, 2, 4, 8, ..., 2^{30} und eine Adresse für den Server mit dem Abstand 2^{31}. Da die Beziehung "Abstand" symmetrisch ist, gibt es für jeden Verweis auf einen Server bei diesem Server einen entsprechenden Rückverweis. Konnte ein Verweis nicht hergestellt werden, weil die entsprechende Nummer noch nicht vergeben wurde, dann müssen zwei Fälle unterschieden werden:

1. Der Abstand ist zu klein, das heißt die Differenz der Servernummern ist zu gering. In diesem Fall wird der Verweis einfach nicht belegt. Sind in der Anfangsphase nur 4 Servernummern vergeben, so ist der geringste Abstand zwischen zwei Servern 2^{30}, die Verweise auf Server mit geringerem Anstand werden zunächst nicht belegt.
2. Der Abstand ist groß genug, aber die Servernummer, auf die verwiesen werden soll, wurde noch nicht vergeben. Diese Verweise müssen belegt werden, da sonst die binäre Suche nicht mehr durchgeführt werden kann. Die Verweise gehen dann jeweils auf den Ersatzserver mit der nächst kleineren, vorhandenen Zahl. In diesem Fall können die

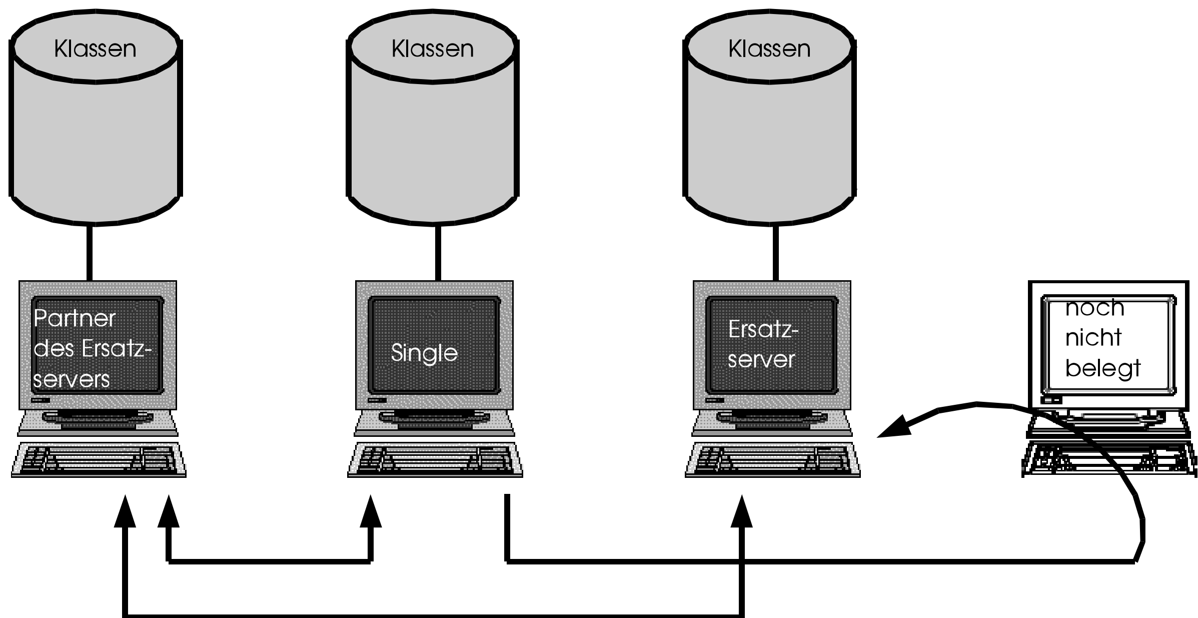

Abbildung 11: Der Verweis auf eine noch nicht vergebene Servernummer richtet sich zunächst auf den Ersatzserver. Der Ersatzserver kann über seinen Partner gefunden werden.

Ersatzserver keinen Rückverweis speichern, da sie bereits einen Partner besitzen. Der Server mit dem »zurechtgebogenen« Verweis ist ein Single (genaugenommen nur ein partieller Single, da er wenigstens einen korrekten Verweis besitzt). Der Partner des Ersatzservers ist genau der Server, der die nächst kleinere Nummer des Single hat. Der Single kann sich also beim Partner des Ersatzservers anmelden und bekommt von ihm alle Informationen über seine(n) Ersatzserver.

Alle unbesetzten oder falschen Verweise werden sofort korrigiert, wenn die entsprechende Nummer an einen Server vergeben wurde. Dabei muß der Server nicht selbst aktiv werden, da der neue Server sich bei seinem Partner meldet und diesen um Eintragung bittet.

4.5.5 Anmelden eines neuen Servers

Im Gegensatz zu Anfragen sind organisatorische Operationen selten und dürfen daher in ihrer Ausführung aufwendiger sein. Allerdings sind auch hier Grenzen zu beachten, damit die beschriebenen Restriktionen eingehalten werden können.

Die Vergabe der Servernummern wurde bislang als ein zentraler Mechanismus betrachtet. Da keine zentralen Komponenten erwünscht sind, soll die Vergabe verteilt ablaufen. Ausgangsbasis ist ein existierendes System und ein neuer Server, der dem System beitreten möchte. Der neue Server kennt allerdings nur die Adresse eines Servers im System.

Zunächst muß der neue Server herausfinden, welche Nummer als nächste vergeben werden soll. Die *Position* eines Servers soll die Anzahl der »älteren« Server bezeichnen. Der Server

kann seine Position durch Intervallschachtelung herausfinden. Dazu wird das Intervall mit [0, 2^{32}-1] initialisiert und an den bekannten Server geschickt. Der befragte Server antwortet mit einem Intervall und einer Serveradresse, an die sich der neue Server weiter wenden soll. Dies wiederholt der neue Server solange bis ein Server, der Ankerserver, anstatt eines Intervalls eine Bestätigung der Registrierung mit der Nummer des neuen Servers schickt. Ab diesem Zeitpunkt ist der neue Server in das System integriert.

Für das obige Anmeldungsverfahren müssen die Server das Intervall anpassen und einen Folgeserver benennen können. Dafür verfügt jeder Server über eine Funktion, die eine Servernummer in die zugehörige Position umrechnet, und deren Umkehrfunktion. Die Untergrenze des Intervalls stellt die kleinstmögliche Position des Servers dar, die Intervallobergrenze analog die größtmögliche Position. Jeder Server kennt seine eigene Servernummer und kann, durch Addition des Abstands, die Servernummern herausfinden auf die er verweist, diese Servernummern wurden folglich alle schon vergeben. Zu jeder Servernummer wird die Position errechnet und das Maximum bestimmt. Die gesuchte Position ist also um mindestens eins größer als dieses Maximum. Ist die Intervalluntergrenze kleiner oder gleich groß wie das ermittelte Maximum, so wird sie auf das Maximum plus eins gesetzt. Die Obergrenze kann analog angepaßt werden, indem die Servernummern aller unbelegten Verweise und Verweise auf Ersatzservern verwendet werden und statt des Maximums das Minimum gebildet wird. Danach grenzt das Intervall die möglichen Positionen des neuen Servers weiter ein. Um die Position möglichst schnell zu finden soll eine binäre Suche durchgeführt werden. Dazu wird der Mittelwert errechnet, indem das arithmetische Mittel aus den beiden Intervallgrenzen gebildet und auf einen ganzzahligen Wert gerundet wird. Die Suche nach einer freien Position soll also beim Mittelwert fortgesetzt werden. Dazu muß zunächst aus der Position die zugehörige Nummer des Folgeservers ermittelt werden. Für die Lage des Folgeservers gegenüber dem Server sind die folgenden drei Fälle denkbar:

1. Es gibt einen direkten Verweis auf den Folgeserver. Dieser Fall kann nicht eintreten, da durch die Existenz des Servers seine Position vergeben ist und diese Position nicht als Mittelwert aus dem Intervall hervorgehen kann.
2. Der Verweis auf den Folgeserver ist nicht eingetragen oder nicht korrekt. Dieser Fall kann ebenfalls nicht eintreten, die Begründung ist analog zum ersten Fall.
3. Der Folgeserver ist nicht durch einen Verweis erreichbar. Dann wird der Server als Folgeserver bestimmt, der dem korrekten Folgeserver am nächsten kommt.

Sobald das Intervall nur noch die Länge eins hat, ist die gesuchte Position gefunden. Der Folgeserver ist bereits der *Ankerserver* (Server mit der nächst kleineren vergebenen Nummer). Der Ankerserver bricht den Suchvorgang ab, reserviert die suchte Position, indem der entsprechende Verweise gleich eingetragen wird. Der neue Server bekommt dann eine Bestätigungsmeldung, in der alle gespeicherten Verweise des Ankerservers mitgeliefert werden.

Die erste Aufgabe des neuen Servers besteht darin, alle seine Verweise soweit möglich zu setzen, dabei ist der Ankerserver hilfreich. Der Ankerserver ist der Server mit der nächst kleineren Nummer und damit der Partner aller Ersatzserver. Wurde in einem Fall der Ersatzserver nicht benötigt, weil der entsprechende Server bereits vorhanden ist, so kennt der Ersatzserver zumindest den korrekten Server. Auf diese Weise kann der neue Server alle seine Verweise setzen und seinen Partnern seine Existenz mitteilen, so daß auch die Rückverweise korrekt belegt werden können.

4.5.6 Abmelden eines Servers

Bei der Abmeldung darf keine Lücke in den Servernummern entstehen. Es kann folglich nur der Server mit der letzten Position abgemeldet werden. Der Server, der sich abmelden möchte, kann mit obigen Verfahren den Server mit der letzten Position ermitteln. Dieser nachrückende Server meldet sich zunächst ab, indem er alle Partner von seinem Wegfall unterrichtet. Die Partner können die Rückverweise entweder löschen oder verwenden den mitgelieferten Ankerserver als Ersatzserver. Erst wenn diese Nachrichten bestätigt wurden, kann der Server mit der letzten Position sicher sein, daß keine Anfragen mehr kommen, und er kann die Daten auf seine beiden nächsten Nachbarn verteilen. Danach lädt der nachrükkende Server die Daten des wegfallenden Servers und übernimmt dessen Rolle. Sobald alle Partner des wegfallenden Servers auf die Adresse des nachrückenden Servers umgestellt haben und dies bestätigen, kann sich der wegfallende Server beenden.

4.5.7 Einführen von Replikaten

Der verteilt Multibaum ist anfällig gegenüber Störungen. Der Ausfall eines Servers kann bereits die Funktion des gesamten Multibaums nachhaltig beeinflussen. Bei einer Netzpartitionierung ist das Funktionieren des Multibaums nur wenig wahrscheinlich.

Zumindest bei einem Rechnerausfall sollten die dort gespeicherten Klassen noch zugänglich sein. Eine Klasse kann nicht mehr auf einem einzigen Rechner gespeichert werden, da sie dann beim Ausfall dieses Rechners nicht mehr verfügbar ist. Daher wird Redundanz bei der Speicherung der Klassen eingeführt. Bislang wurde sichergestellt, daß die Namensintervalle, für die ein Server zuständig ist, lückenlos aneinanderstoßen. Dieses Zuständigkeitsintervall eines Servers soll als dessen *Originalintervall* bezeichnet werden. Ein Server hält als Redundanz die Klassen, die in die Originalintervalle seiner beiden Nachbarn fallen. Bei dessen Ausfall muß der Multibaum umgestaltet werden, als ob der Server sich abgemeldet hätte. Die Klassen des Originalintervalls kann dem ersetzenden Server von einem seiner Nachbarn übergeben werden. Durch die doppelte Redundanz einer Klasse gehen keine Daten verloren, wenn beliebig viele, aber nicht mehr als zwei benachbarte Rechner ausfallen. Ohne Frage ist jedoch der Aufwand um den Multibaum zu reparieren gewaltig.

Als erfreulicher Nebeneffekt der Redundanz kann das Suchverfahren weiter beschleunigt werden, da ein Server nicht nur Auskunft über sein Originalintervall, sondern auch über das seiner Nachbarn geben kann. Der Nachrichtenaufwand bei einer Suche kann dadurch um mindestens eine Nachricht verringert werden, da der letzte Suchschritt mit der Intervallänge 1 nicht mehr vollzogen werden muß.

4.6 Vergleich der Basismechanismen

Es wurden verschiedene Basismechanismen vorgestellt, die unterschiedliches Verhalten zeigen. Es sollen deshalb die Eigenschaften der Basismechanismen in den Bereichen Netzbelastung, Antwortzeit, Fehlerverhalten, Lastbalancierung, Ressourcenverbrauch und Einschränkungen zusammenfassend dargestellt werden.

4.6.1 Skalierbarkeit

Skalierbarkeit ist kein zweiwertiges Prädikat, das entweder erfüllt oder nicht erfüllt ist, vielmehr gibt es unterschiedliche Grade der Skalierbarkeit. Für die Skalierbarkeit des Basismechanismus ist auch der Aufwand für das An- und Abmelden von Servern entscheidend. Als Aufwand muß das Aufkommen an Nachrichten in Zahl und Größe sowie der Aufwand zur Implementierung berücksichtigt werden.

Skalierbarkeit ist erreicht, wenn im laufenden Betrieb neue Server integriert werden können. *Volle Skalierbarkeit* ist zusätzlich erreicht, wenn durch das Abmelden einer beliebigen Zahl von Servern alle Daten noch verfügbar sind. Im Extremfall werden alle bis auf einen Server abgemeldet, dann müssen alle Daten auf dem verbleibenden Server gespeichert werden.

Zunächst sollen die Mechanismen betrachtet werden, die eine volle Skalierbarkeit ermöglichen. Darunter fällt der Multicast, die komplette Namensauflösung mit dem DNS und der verteilte Multibaum.

Das Hinzufügen von neuen Servern ist bei diesen Mechanismen trivial oder wurde bereits beschrieben. Das Beenden eines Servers erfordert jedoch organisatorische Änderungen in der verteilten Struktur der Informationsausbreitung. Beim Multicast (sofern als spannender Baum organisiert) und beim verteilten Multibaum müssen die Strukturen umgebaut werden, dies erfordert Zeit, für die der Mechanismus (in Teilen) nicht zur Verfügung steht. Das Synchronisieren der Benutzerzugriffe mit den organisatorischen Zugriffen ist in diesen verteilten Systemen nicht trivial, insbesondere dann, wenn zudem noch Ausfälle oder anderes

Fehlverhalten berücksichtigt werden muß. Die Verwendung des DNS löst die Probleme nicht, sondern verlagert sie in ein bereits bestehendes System. Da beim Abmelden der Server seine kompletten Daten auf einen anderen Server übertragen muß, ist die Belastung des Netzwerks zum Erreichen der vollen Skalierbarkeit enorm, die organisatorische Nachrichten nehmen sich dagegen gering aus.

Bei den beheimateten Klassen führt das Beenden von Servern zum Verlust der Verfügbarkeit der dort gespeicherten Klassen. Die vorgeschlagene Lösung der replizierten Klassen auf den Sekundärservern kann die Wahrscheinlichkeit der Nichtverfügbarkeit senken, aber nicht ausschließen. Der Verzicht auf volle Skalierbarkeit führt in diesem Fall zu einer Skalierbarkeit ohne zusätzlichem Aufwand.

Das Verfahren »bestätigtes Propagieren« bezieht sich immer auf eine paarweise Verbindung von Servern, es ist damit ohne Aufwand beliebig skalierbar. Das Abmelden von Servern ist jedoch nur dann möglich, wenn sich keine Agenten mehr im Wirkungskreis des Servers befinden.

Soll die Zahl der Server nur als wachsend vorausgesetzt werden, so sind die Mechanismen mit einfacher Skalierbarkeit ausreichend. Muß jedoch berücksichtigt werden, daß Server nur zeitweise betrieben werden, so ist ein Mechanismus mit voller Skalierbarkeit zu wählen.

4.6.2 Netzbelastung

Die Netzbelastung eines Basismechanismus zu untersuchen, ist aufwendig und hängt von vielen äußeren Umständen ab. Aus diesem Grund soll hier als Netzbelastung nur die Anzahl der Nachrichten in Abhängigkeit von der Anzahl Server bei einer Anfrage betrachtet werden.

Beim Multicast entstehen die meisten Nachrichten, da fast alle Server eine Nachricht bekommen. Selbst wenn von einer hohen Lokalität der Nachrichten ausgegangen werden kann, so ist die Netzbelastung durch die große Anzahl Nachrichten dennoch beachtlich. Im Gegensatz dazu werden bei den beheimateten Klassen und dem bestätigten Propagieren jeweils nur eine Nachricht benötigt. Die Strukturen, die auf Bäumen basieren, also DNS und der verteilte Multibaum, benötigen eine logarithmische Anzahl an Nachrichten. In der Tabelle 8 ist zu jedem Basismechanismus der Aufwand an Nachrichten pro Anfrage aufgeführt.

Basismechanismus	**Mittlere Anzahl Nachrichten pro Anfrage bei insgesamt n Servern**
Bestätigtes Propagieren	O (1)
Multicast	O (n)
Beheimatete Klassen	O (1)

Basismechanismus	**Mittlere Anzahl Nachrichten pro Anfrage bei insgesamt n Servern**
DNS (komplette Namensauflösung)	O (log n)
Beheimatete Klassen und DNS	O (1)
Verteilter Multibaum	O (log n)

Tabelle 8: Zusammenfassung der Netzbelastung

Da das bestätigte Propagieren das gestellte Problem nicht vollständig lösen kann, ist die Netzbelastung von den verbleibenden Mechanismen bei den beheimateten Klassen am geringsten.

4.6.3 Antwortzeit

Die Antwortzeit ist der Zeitbedarf um eine angeforderte Klasse auszuliefern, oder zu bestätigen, daß diese Klasse nicht in der Codebasis enthalten ist.

Bei allen Basismechanismen mit Ausnahme des Multicasts werden die Nachrichten zeitlich nacheinander abgesetzt, insofern die die Antwortzeit bei diesen Verfahren analog zu der Netzbelastung zu sehen. Allerdings muß noch die mittlere Entfernung (als Verzögerung) der Nachrichten berücksichtigt werden. Die Entfernung der Nachrichten bei den Verfahren ist nicht beschränkt es handelt sich (mit Ausnahme des DNS) jeweils um globale Kommunikationen.

Der Multicast bildet bezüglich der Antwortzeit eine Ausnahme, da der Algorithmus parallel abläuft. Wird die Information entlang eines spannenden Baums mit festem Rang verbreitet, dann ist die Anzahl der Nachrichten zwar linear, die Antwortzeit wächst dagegen nur logarithmisch. Nur bei einem als Kette entarteter spannender Baum ist die Antwortzeit und die Anzahl der Nachrichten linear. Bei einem spannenden Baum mit nur einem inneren Knoten (der Knoten bildet die Mitte eines Sterns) ist die Antwortzeit sogar konstant. Die Antwortzeit beim Multicast ist folglich stark von der Struktur des spannenden Baums abhängig. Die geschilderten Extremfälle (Kette und Stern) sollen hier nicht als typisch angesehen werden. Bei einem Baum mit beliebigem aber festem Rang ist die Antwortzeit logarithmisch zur Anzahl der Server.

Die kürzeste Antwortzeit weist das Verfahren der beheimateten Klassen und das bestätigte Propagieren auf.

4.6.4 Fehlerverhalten

Unter dem Begriff Fehlerverhalten soll nicht nur betrachtet werden, ob ein Mechanismus gegen Störeinflüsse geschützt werden kann, sondern auch welcher Aufwand dafür notwendig ist.

Der Rechnerausfall unterscheidet sich nur dadurch vom Beenden eines Servers, daß der Server keinen Systemabschluß durchführen und andere Server nicht unterrichten kann. Für das bestätigte Propagieren und die beheimateten Klassen ist keine Unterscheidung notwendig, da bei diesen Verfahren ohnehin kein Systemabschluß im Netzwerk durchgeführt wird. Bei der Implementierung des Multicasts und des verteilten Multibaums muß ein Rechnerausfall (oder eine beliebige Anzahl von Rechnerausfällen) zunächst erkannt werden, die Reorganisation der Baumstruktur müßte von einem zu bestimmenden Server übernommen werden. Die Schwierigkeiten liegen nicht allein in der Ausfallerkennung und der Reorganisation, sondern auch in der sicheren Synchronisation der Algorithmen unter möglichst geringer Einschränkung der Verfügbarkeit des Basismechanismus.

Nicht nur die Rechner können Fehlerquellen darstellen, das Netzwerk selbst kann ebenfalls für eine Reihe von Fehlerfällen sorgen. Ausgehend von einer sicheren Verbindung soll hier nur die Auswirkung einer Netzpartitionierung auf die verschiedenen Basismechanismen dargestellt werden.

Die Verfahren, die nur eine Verbindung pro Abfrage benötigen (beheimatete Klassen und bestätigtes Propagieren), arbeiten auch bei Netzpartintionierung, wenn beide Server in der gleich Partition liegen. Alle Verfahren, die auf (Bäume oder ähnliche) Strukturen beruhen, sind anfälliger, da alle Server des Pfades vom Ausgangsserver zum Zielserver in der gleichen Partition liegen müssen.

4.6.5 Lastbalancierung

Wie bereits bei den Eigenschaften der Klassen angemerkt, ist die Zugriffshäufigkeit auf verschiedene Klassen unterschiedlich. Damit nicht ein Server überlastet wird, während ein anderer Server nur selten Anfragen erhält, ist eine Balancierung der Lasten sinnvoll.

Eine Lastbalancierung ist bei den voll skalierbaren Mechanismen ohne Probleme möglich. Beim bestätigten Propagieren ist eine Lastbalancierung überhaupt nicht möglich. Bei den beheimateten Klassen kann durch replizieren von Klassen eine Verteilung erreicht werden, da diese aber nicht nach der dynamischen Last gesteuert werden kann, ist dieser Basismechanismus nicht für eine Lastbalancierung geeignet.

4.6.6 Ressourcenverbrauch

Die Preisentwicklung der externen Speichermedien (Festplatten) ist seit einigen Jahren stark fallend. Der Ressourcenverbrauch der einzelnen Mechanismen ist vor diesem Hintergrund kein bedeutendes Kriterium, dennoch soll der Ressourcenverbrauch nicht verschwiegen werden.

Beim bestätigten Propagieren kann eine Klasse erst gelöscht werden, wenn sich keine Agenten, die diese Klasse verwenden, im Einflußbereich des Codeservers befinden. Durch diese Forderung kann eine Klasse notwendigerweise auf vielen Codeservern gleichzeitig gespeichert sein.

Alle anderen Mechanismen gehen davon aus, daß die Klasse mindestens einmal gespeichert ist. Replikate sind wünschenswert, aber nicht verpflichtend.

4.6.7 Einschränkungen

Jeder Basismechanismus geht von einer Reihe von zu erfüllenden Annahmen aus. Manche dieser Annahmen stellen für den Gebrauch Einschränkungen dar, andere Annahmen sind ohne Problem zu erfüllen.

Der Mechanismus der beheimateten Klassen, beruht auf dem Umbenennen von Klassen, dies muß das Java Laufzeitsystem zulassen. Außerdem muß der Agentenprogrammierer sich an die neuen Namenskonventionen halten, wenn er eine Klasse verwenden will.

Der verteilte Multibaum setzt voraus, daß sich die Klassen beliebig auf die Server verteilen lassen, dafür ist die Kooperation der Betreiber der Server notwendig. Der Betreiber eines Servers stellt nur Ressourcen zur Verfügung, kann aber nicht beeinflussen, von wem diese Ressourcen in welcher Weise genutzt werden. Während die Benutzer bei den beheimateten Klassen genau wissen, wo ihre Klassen gespeichert sind und ein Interesse an der Verfügbarkeit des Servers besitzen, ist beim verteilten Multibaum ein selbstloses Menschenbild vorausgesetzt. Eine Befragung der Beteiligten am Projekt »Mole« ergab, daß diese Voraussetzung nicht zur Bedingung für ein funktionierendes System werden darf. Der verteilte Multibaum ist technisch sicherlich interessant, ist in der Praxis jedoch ohne entsprechende Organisation (zum Beispiel ein einziger Betreiber aller Codeserver) nicht zu betreiben.

4.7 Implementierter Mechanismus

Vorgesehen war zunächst die Implementierung mehrerer der vorgestellten Mechanismen. Die beschriebenen Eigenschaften einiger Mechanismen zeigten jedoch, daß diese nicht zur Lösung geeignet sind. So kam der Multicast aufgrund der großen Nachrichtenanzahl nicht in Frage, das bestätigte Propagieren löst die dynamischen Referenzen nicht und die komplette Namensauflösung mit dem DNS wird wegen geringer Eignung und organisatorischer Schwierigkeiten nicht verwendet.

Als Auswahl blieben nur noch die beheimateten Klassen (mit oder ohne DNS) und der Multibaum übrig. Der hohe Aufwand zur Implementierung und die Ungewißheit über die Kooperationsfähigkeit der Betreiber ließen auch den Multibaum für eine Implementierung ungeeignet erscheinen.

Zur Implementierung wurde daher der Basismechanismus »Beheimatete Klassen « ausgewählt. Es ist weder üblich Internet-Adressen zu verwenden noch erlaubt die Sprache Java den Aufbau einer Verbindung ohne Angabe eines DNS-Namens. Aus diesem Grund ist es sinnvoll die Kombination »Beheimatete Klassen und DNS« umzusetzen.

5 Verbesserung der Performanz

Eine Steigerung des Durchsatzes und die Verbesserung der Antwortzeit stehen in direktem Konflikt zum Ressourcenverbrauch. Da der zur Verfügung stehende Platz an Speicher pro Server begrenzt ist, muß eine Strategie entwickelt werden, die entscheidet, ob eine Klasse für weitere Zugriffe gespeichert oder sogar präventiv besorgt und lokal gespeichert werden soll. Der zur Verfügung stehende Platz soll so eingesetzt werden, daß ein möglichst großer Gewinn an Performanz erzielt wird.

Neben dem Basismechanismus gibt es daher einen weiteren, schnellen Dienst, der für die Beschaffung der Klassen sorgt. Während beim Basismechanismus die Verfügbarkeit und Zuverlässigkeit die wichtigsten Anforderungen waren, steht die Antwortzeit an oberster Stelle beim schnellen Dienst. Die beiden Dienste (Basismechanismus und schneller Dienst) sind getrennt voneinander zu betrachten, die benötigten Ressourcen werden unabhängig voneinander eingeteilt.

Zur Verbesserung der Performanz sollen drei unterschiedliche Strategien verfolgt werden:

1. Wiederverwendung
   Ein Server kann und sollte mehrere Engines bedienen. Folglich ist es möglich, daß die gleiche Klasse von mehreren Engines angefordert wird. Der Server könnte diese wiederholten Anfragen schneller beantworten, wenn er die Klassen in einem Zwischenspeicher lokal hält.
2. Ausnützen schneller Verbindungen
   Kann ein Server andere Server ausfindig machen, zu denen eine leistungsfähige, schnelle Verbindung besteht, dann können beide Server durch Kooperationen den eigenen Ressourcenverbrauch senken und so die Gesamtperformanz steigern.
3. Vorratshaltung
   Die Server sollen nicht nur auf Anforderung aktiv werden, am schnellsten kann eine Anfrage bedient werden, wenn die gewünschte Klasse bereits präventiv besorgt wurde.

Diese drei Teilaspekte sollen in den folgenden Kapiteln näher betrachtet werden.

5.1 Der Zwischenspeicher

Es steht eine gewisse Menge an Speicher zur Verfügung. Bei der Speicherung von Daten müssen möglicherweise andere Daten aus dem Speicher entfernt werden. Diese Problem ist nicht einmalig, es tritt in ähnlicher Weise bei Betriebssystemen in der Verwaltung von virtuellem Speicher auf. Einige Algorithmen zur Seitenersetzung sind in [Tanenbaum 1992] aufgeführt. Primär interessiert daher, ob es möglich ist einen dieser Algorithmen hier einzusetzen.

Analog zu Seiten beim virtuellen Speicher werden beim Zwischenspeicher Klassen gespeichert. Das Laden einer Seite vom Sekundärspeicher entspricht dem Übertragen einer Klasse über das Netzwerk. Beim Auslagern einer Klasse muß die Klasse zwar nicht zurück übertragen werden, da sie nicht geändert werden darf, aber es kann sinnvoll sein, Informationen über das Zugriffsverhalten auf die Klasse zurück zu übertragen. Alle Vor- und Nachteile von Algorithmen der Seitenersetzung gelten folglich entsprechend.

Das vorliegende Problem ist jedoch komplizierter als die Verwaltung von virtuellem Speicher. Der Zeitbedarf für das Auslagern und Einlesen einer Seite wird im Betriebssystem für alle Seiten als ungefähr gleich hoch angenommen. Diese Annahme trifft beim Codeserver nicht zu, da die Zeit zum Laden einer Klasse von ihrer Größe und der Entfernung ihrer Quelle abhängt. Die sich daraus ergebenden zeitlichen Unterschiede können durchaus dazu führen, daß es günstiger ist, eine kleine Klasse mehrmals zu laden, wenn dadurch eine große Klasse gehalten werden kann.

Die Problematik wird bei der optimalen Ersetzungsstrategie deutlich. Bei der optimalen Seitenersetzung genügt es einfach die Seite auszulagern, die erst in weiter Zukunft wieder benötigt wird. Dem optimalen Klassenersetzung-Algorithmus genügt dieses einfache Kriterium nicht. Die zu optimierende Größe ist der Zeitbedarf, deshalb gibt die Kostenfunktion f für eine Klasse die benötigte Zeit zum Auslagern und Nachladen an. Eine Folge von zu ersetzenden Klassen $k_1, k_2, \ldots, k_n$ ist optimal, wenn die Summe der Kosten $f(k_1)+f(k_2)+\ldots+f(k_n)$ minimal wird. Nach dieser Definition kann es mehr als eine optimale Folge geben. Selbst wenn die zukünftigen Klassenanfragen bekannt sind, so ist die Suche nach einer optimalen Folge dennoch aufwendig. Es besteht großen Ähnlichkeit mit dem Rucksackproblem, daher ist zu vermuten, daß auch dieses Problem NP vollständig ist. Die zukünftigen Zugriffe sind jedoch nicht bekannt, es kann deshalb auf den Nachweis der NP Vollständigkeit an dieser Stelle verzichtet werden.

Noch vor der Untersuchung eines speziellen Algorithmus scheint es zunächst offensichtlich, daß es besser ist, eine kleine Klasse auszulagern als eine große Klasse. Aber das ist so nicht richtig, da der Nutzen durch das Auslagern der kleinen Klasse geringer ist. Sollte es nötig sein, zwei kleine Klassen auszulagern, dann können diese insgesamt eine größere Datenmenge darstellen als die zunächst größere Klasse.

Die Klassen werden im Dateisystem angelegt. Die Summe der Dateigrößen darf nicht das festgelegte Limit an Speicherplatz übersteigen. Um den Speicherplatz immer optimal zu nützen, könnte man nun vorschlagen, immer die Menge von Klassen auszulagern, so daß nach dem Speichern der neuen Klasse möglichst wenig Speicherplatz frei bleibt. Der sparsame Umgang mit dem Speicherplatz ist an dieser Stelle nicht angebracht, da das beschriebene Problem NP-Vollständig ist (vgl. Rucksackproblem). Es wird hier vorausgesetzt, daß die Größe aller Klassen relativ zum insgesamt verfügbaren Speicherplatz sehr klein ist, dadurch ist es unbedeutend, wie groß der verschenkte Speicherplatz ist. Stehen beispielsweise 10 MB Speicherplatz zur Verfügung, dann sind 50 KB, die nicht benutzt werden, sicherlich kein relevantes Problem, da diese gerade einem halben Prozent entsprechen.

Durch die obige Feststellung kann der folgende einfache Auslagerungsalgorithmus sinnvoll eingesetzt werden: die Klassen sind, gemäß Ihrem Wert der Bewertungsfunktion, aufsteigend sortiert. Soll eine Klasse (vorübergehend) gespeichert werden, so wird die Bewertungsfunktion für diese Klasse berechnet und die Klasse entsprechend dem Wert einsortiert. Bevor die Klasse jedoch gespeichert werden kann, werden die Klassen von oben beginnend ausgelagert, bis genug freier Speicher vorhanden ist. Sollte dabei die Klasse selbst ausgelagert werden, so wird sie nicht zwischengespeichert.

Die bekannten Algorithmen zur Seitenauslagerung können folgendermaßen zusammengefaßt werden. Eine Funktion g gibt den Wert einer Seite an, das heißt, wie hoch die Erwartung ist, daß diese Seite noch benötigt wird. Eine der Seiten mit der kleinsten Bewertung wird ausgelagert. Die verschiedenen Algorithmen unterscheiden sich nur durch die Bewertungsfunktion und die Implementierung des Verfahrens. Die Tabelle 9 zeigt die Bewertungsfunktionen für einige bekannten Algorithmen.

Name des Algorithmus	Bewertungsfunktion
FIFO (first in first out)	G = Zeitpunkt des Einladens als Ganzzahl
NRU (not recently used)	G = 1, wenn Zugriff im letzen Zeitabschnitt, sonst G = 0
Second-Chance	G = 0, wenn kein Zugriff in den letzen beiden Zeitabschnitten, G = 1, wenn kein Zugriff im letzten Zeitabschnitt, sonst G = 2, eine feste Nachkommastelle legt eine Reihenfolge der Seiten fest.
LRU (least recently used)	G = Zeitpunkt des letzten Zugriffs als Ganzzahl
NFU (not frequently used)	G = Zahl der Zugriffe
Random	G = beliebige Zufallszahl

Tabelle 9: Bekannte Algorithmen zur Seitenersetzung nach [Tanenbaum 1992]

Die bekannten Algorithmen berücksichtigen also nur die Zeit des letzten Zugriffs oder Einladens; darauf aufbauend soll ein geeigneter Algorithmus für den Zwischenspeicher des Codeservers entwickelt werden. Es können jedoch auch andere Faktoren in der Bewertungs-

funktion berücksichtigt werden. Für jede Einflußgröße der folgenden Liste wurden die korrespondierenden Voraussetzungen angegeben.

- Zeitpunkt des Einladens
  Hier wird vorausgesetzt, daß eine erst vor kurzer Zeit geladene Klasse noch häufig benötigt wird, während eine Klasse, deren Erstnachfrage lange zurückliegt, voraussichtlich nicht mehr benötigt wird.
- Zeitpunkt des letzten Zugriffs
  Es wird vorausgesetzt, daß ein Zugriff unwahrscheinlicher wird, je länger der letzte Zugriff zurückliegt.
- Anzahl der Zugriffe
  Wird die absolute Zahl der Zugriffe berücksichtigt, so geht man davon aus, daß häufige Zugriffe in der Vergangenheit auch häufige Zugriffe in der Zukunft erwarten lassen.
- Anzahl der Zugriffe in einem Zeitabschnitt
  Gehen die Zugriffe pro Zeiteinheit ein, dann wird eine zeitlich gleichmäßige Verteilung der Zugriffe auf die Klasse vorausgesetzt.
- Größe der Klasse
  Die Größe der Klasse kann berücksichtigt werden, da sich die Verzögerung bei kleinen Klassen stärker auswirkt als bei großen Klassen.
- Dauer für die Übermittlung der Klasse
  Die Dauer zur Besorgung der Klasse impliziert, daß es vorzuziehen ist, eine schnell verfügbare Klasse auszulagern, als eine Klasse, deren Übertragung viel Zeit in Anspruch nimmt.
- Anzahl zu erwartender Zugriffe
  Ist die Anzahl der zu erwartenden Zugriffe durch Heuristiken bekannt, so wird bei deren Beachtung von einem raschen Eintreffen der heuristischen Prognose ausgegangen.

Eine prinzipielle Einschränkung gibt es nicht, jede einzelne Voraussetzung kann in einem Agentensystem erfüllt sein. Tatsächlich sind aber zur Zeit noch keine typischen Verhaltensmuster für die Agenten und damit auch nicht für die Anfragereihenfolge bekannt[6]. Es sollen

6 Messungen am Agentensystem »Mole« können erst nach der Umstellung auf Java 1.1 erfolgen, da unter Java 1.0 keine geladenen Klassen aus dem Speicher entfernt werden. Die Umstellung von Mole auf Java 1.1 wird erst nach Abgabe dieser Arbeit erfolgen. Bei einer Messung kann das neue Verhalten der Virtal Maschine nicht identisch nachempfunden werden, da nicht bekannt ist, wann eine nicht mehr benötigte Klasse entfernt wird. Dieses Verhalten beeinflußt jedoch entscheidend die Messung, beispielsweise sollen von einer Klasse eine große Menge von Objekte erzeugt werden. Ist immer mindestens ein Objekt der Klasse vorhanden, so ist die Klasse offensichtlich nicht überflüssig. Wird allerdings immer nur ein Objekt erzeugt und vor der Erzeugung eines weiteren Objekts wieder gelöscht, dann ist es möglich, daß die Klasse vor jeder Erzeugung aus dem Speicher entfernt wurde und neu besorgt werden muß. Eine Nachbildung dieses Verhaltens würde immer ein bestimmtes Modell voraussetzen, dadurch entfällt die Messung, da bereits das Modell die Anfragereihenfolge bestimmt.
Selbst wenn eine realistische Umgebung Messungen zuließe, so müßte noch eine Mischung typischer Agentenanwendungen zusammengestellt werden, über typische Agentenanwendungen läßt sich zur Zeit jedoch nur spekulieren.

daher möglichst wenig Voraussetzungen angenommen werden, deren Erfüllung nicht nachgewiesen werden kann. Die Entfernung von Klassen aus dem Zwischenspeicher muß nicht so extrem zeitkritisch erfolgen, wie es beim Seitenersetzungsalgorithmus in Betriebssystemen notwendig ist.

Als Grundlage für den Algorithmus soll hier der LRU (least recently used) und Random untersucht werden.

5.1.1 LRU (least recently used)

LRU beruht auf der Überlegung, daß eine lange nicht benötigte Klasse auch in Zukunft wohl nicht mehr benötigt wird. Es wird folglich die Klasse aus dem Zwischenspeicher entfernt, auf die am längsten nicht mehr zugegriffen wurde. Sind Angaben darüber vorhanden, daß noch Zugriffe auf die Klasse erwartet werden, so wird pro erwartetem Zugriff ein konstanter Anteil auf den Zeitpunkt des letzten Zugriffs addiert. Der Zeitpunkt des letzten Zugriffs wird dadurch, anschaulich gesehen, auf einen Zeitpunkt in der Zukunft gesetzt. Das deckt sich mit der Idee, daß diese Zugriffe erst noch erwartet werden. Finden Zugriffe auf die Klasse statt, dann wird der Zeitpunkt des letzten Zugriffs aktualisiert, aber auch die Anzahl der erwarteten Zugriffe vermindert.

Das Verfahren soll letztlich eine Aussage treffen, wie wahrscheinlich die Klasse noch benötigt wird. Durch die Wahl des LRU als Grundlage wird folgende Annahme vorausgesetzt:

Je länger der Zeitpunkt des letzten Zugriffs auf die Klasse zurückliegt,
desto unwahrscheinlicher wird ein weiterer Zugriff auf die Klasse.

Die Wahrscheinlichkeit für einen weiteren Zugriff auf eine Klasse kann als Funktion in Abhängigkeit von der Dauer des letzten Zugriffs, angegeben werden. Leider ist diese Funktion weder bekannt, noch waren Angaben darüber in der Literatur zu finden. Dies verwundert nicht weiter, wenn man bedenkt, daß dieser Zusammenhang stark von der jeweiligen Anwendung abhängt. Werden für die Agenten einer Anwendung die Klassen nur für kurze Zeit stark nachgefragt, dann geht die Wahrscheinlichkeit nach einer kurzen Zeit auf Null zurück, die Funktion müßte also stark abfallen. Werden in einer anderen Anwendung die Klassen allerdings selten, aber über einen längeren Zeitraum nachgefragt, dann wäre die Wahrscheinlichkeit für alle Zeiträume nahezu konstant, der Funktionsverlauf flach. Das Agentensystem ist so flexibel, daß es beide Arten von Anwendungen unterstützt. Es könnte nun beispielsweise zunächst hauptsächlich der erste Anwendungsfall auftreten, nach einer gewissen Zeit jedoch nur noch der Zweite. Der ideale Verlauf der Wahrscheinlichkeitskurve ist folglich von den Anwendungen abhängig und damit sicherlich nicht zeitlich invariant.

Um obigen Überlegungen gerecht zu werden, soll auf eine Vorgabe der Wahrscheinlichkeitsfunktion verzichtet werden. Vielmehr soll das System diese Funktion selbst ermitteln und aus der aktuellen Situation die Funktion laufend anpassen. Um diese Dynamik zu erreichen, wird von dem Gedanken einer stetigen Wahrscheinlichkeitsfunktions abgerückt, und

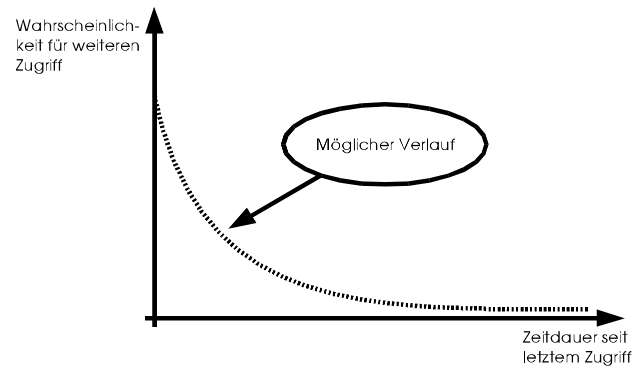

Abbildung 12: Möglicher Verlauf der Wahrscheinlichkeitsfunktion nach dem LRU-Ansatz

es wird eine diskrete Funktion partiell definiert. Die Zeitachse wird in eine feste Anzahl von Zeitintervallen unterteilt. Das erste Intervall beginnt bei Null, alle anderen Intervalle setzen am vorhergehenden Intervall an. Die Intervalle enden auf den Zweierpotenzen, also t_0, $t_1 = 2\ t_0$, $t_2 = 4\ t_0$, $t_3 = 8\ t_0$, ..., $t_n = 2^n\ t_0$. Das letzte Intervall I_n bleibt nach oben offen. Von der Zeitdifferenz Δt kann auf das Intervall I_x geschlossen werden, indem nur der ganzzahlige Anteil x des Logarithmus der Zeitdifferenz Δt berechnet wird. Zu jedem Intervall I_x gibt es einen Zähler Z_x, sowie einen Gesamtzähler Z.

Die Anzahl der Intervalle n und die kleinste Intervallänge t_0 kann fest gewählt sein oder dynamisch ermittelt werden. Die beiden Parameter sollten so gewählt werden, daß der Zähler der Intervalle I_0 und I_{n-1} stets größer als Null und der Zähler des Intervalls I_n stets Null ist. Werden die Parameter fest eingestellt, so sollten dies so gewählt werden, daß zumindest die Bedingungen $Z_0 > 0$ und $Z_n = 0$ erfüllt bleiben.

Bei jedem Zugriff wird die Zeitdifferenz Δt zum letzten Zugriff errechnet. Werden die erwarteten Zugriffen mit berücksichtigt, dann ergibt sich folgende Formel:

$$\Delta t = \max\left(0, t_{jetzt} - \left(t_{letzter\ Zugriff} + t_{Bonus}\ Z_{erwartet}\right)\right)$$

Wobei t_{Bonus} die Zeitspanne ist, in der ein erwarteter Zugriff erfolgt. Mit der berechneten Zeitdifferenz Δt kann das entsprechende Intervall bestimmt werden: a := X(Δt) = $\log_2(\Delta t)$. Der entsprechende Zähler Za und der Summenzähler Z werden jeweils um eins erhöht.

Die Wahrscheinlichkeit, daß auf eine Klasse nochmals zugegriffen wird, kann für jedes Intervall wie folgt berechnet werden:

$$x = \mathrm{trunc}\left(\mathrm{ld}\left(\Delta t\right)\right)$$

$$\mathrm{W}(x) = \frac{\sum_{i=x}^{n} Z_i}{Z}$$

Die Abbildung 13 zeigt einen möglichen Verlauf der so definierten Wahrscheinlichkeitsfunktion. Bei jedem Zugriff wird die Wahrscheinlichkeitsfunktion aktualisiert, da einmal registrierte Zugriffe in den Ständen der Zähler immer Berücksichtigung finden, reagiert das Verfahren mit der zunehmenden Zugriffen immer langsamer auf Veränderungen. Es ist daher ratsam eine Alterung der Zählerstände einzuführen: erreicht der Summenzähler Z einen bestimmten Schwellenwert, dann werden die Werte aller Zähler halbiert, und die Summe wird neu berechnet, damit sich Rundungsfehler nicht aufsummieren können.

Letztlich bekommt man mit obigem Verfahren die relative Wahrscheinlichkeit, mit der zu erwarten ist, daß ein weiterer Zugriff auf die Klasse erfolgt. Die absolute Wahrscheinlichkeit kann das Verfahren nicht liefern, da nur Zugriffe in die Statistik eingehen, die Klassen

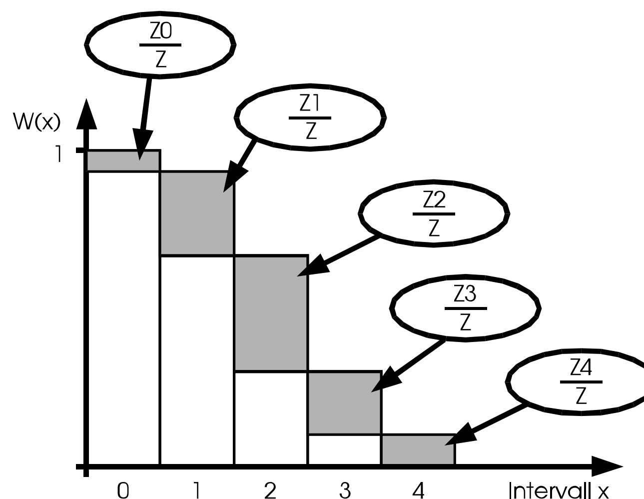

Abbildung 13: Diskrete Wahrscheinlichkeitsfunktion als Beispiel mit 5 Intervallen

betreffen, die sich noch im Zwischenspeicher befanden. Aber auch mit der relativen Wahrscheinlichkeit ließe sich bereits ein Algorithmus implementieren, der dem LRU-Algorithmus entspricht.

Es soll jedoch noch der Zeitbedarf für das Übertragen der Klasse berücksichtigt werden. Dazu wird zunächst die Übertragungsrate Ü als Quotient der Größe einer Klasse durch deren Übertragungszeit berechnet. Die Bewertungsfunktion soll für eine hohe Übertra-

gungsrate einen kleineren Wert und für kleine Übertragungsraten einen großen Wert ergeben. Der folgende Term gibt die zu erwartende Zeitersparnis S pro Byte an, wenn die Klasse im Zwischenspeicher verbleibt:

$$\mathrm{S}(K) = \frac{\mathrm{W}(\mathrm{X}(K))}{\ddot{\mathrm{U}}(K)} \qquad \textit{mit} \quad \ddot{\mathrm{U}}(K) = \frac{\mathrm{Größe}(K)}{\ddot{\mathrm{U}}\text{-}\mathrm{Zeit}(K)}$$

Wobei die Funktion W die Wahrscheinlichkeit für einen weiteren Klassenzugriff (siehe oben) zu einem Zeitintervall berechnet, die Funktion X das Zeitintervall für eine Klasse K ermittelt (siehe oben) und die Funktionen Größe und Ü-Zeit die Größe beziehungsweise die Zeitdauer für das Besorgen[7] einer Klasse K zurück geben.

Um möglichst viel Zeit beim Beantworten von Klassenanfragen sparen zu können, soll nun die Summe der Zeitersparnisse aller gespeicherten Bytes maximiert werden. Die kann einfach dadurch geschehen, daß die Klassen aus dem Zwischenspeicher entfernt werden, deren Zeitersparnis pro Byte - genau diesen Wert berechnet die Funktion S - am geringsten ist. Folglich ist die Funktion S die gesuchte Bewertungsfunktion.

Trifft die LRU-Voraussetzung von Seite 63 nicht zu, dann kann der LRU-Algorithmus nicht eingesetzt werden. Eine gute Alternative stellt der Random-Algorithmus dar.

5.1.2 Random-Algorithmus

Praktische Untersuchungen bei der Seitenersetzung in Betriebssystemen ergaben durchaus gute Ergebnisse, wenn die zu ersetzende Seite zufällig ausgewählt wird. Für den Einsatz des Random-Algorithmus im Codeserver muß der Algorithmus angepaßt werden, um die Kosten der Übertragung der Klasse im Netzwerk berücksichtigen zu können.

Bei der Auswahl der Klasse, die auszulagern ist, wird nicht jede Klasse mit gleicher Wahrscheinlichkeit berücksichtigt, statt dessen wird der Kehrwert der Übertragungsrate verwendet:

$$\mathrm{R}(K) = \frac{1}{\ddot{\mathrm{U}}(K)} = \frac{\ddot{\mathrm{U}}\text{-}\mathrm{Zeit}(K)}{\mathrm{Größe}(K)}$$

Die Funktion R gibt die relative Wahrscheinlichkeit einer Klasse ausgelagert zu werden an. Um die absolute Wahrscheinlichkeit zu bekommen, muß die relative Wahrscheinlichkeit wie folgt normiert werden:

7 Da mit der Formel die zu erwartende Zeitersparnis errechnet werden soll, müßte auch die zu erwartende Zeitdauer für die Besorgung eingesetzt werden. Diese ist jedoch nicht bekannt, da hier nicht von Netzwerkverbindungen mit garantierten Qualitäten ausgegangen wird. Bei der Abschätzung des erwarteten Zeitbedarf wird selbstverständlich auch die Antwortzeit (Netzverzögerung) berücksichtigt (siehe Seite 72 ff).

$$\mathsf{A}(K) = \frac{\mathsf{R}(K)}{\sum\limits_{k \in Speicher} \mathsf{R}(k)}$$

Soll eine neue Klasse im Zwischenspeicher angelegt werden, dann werden solange Klassen ausgelagert, bis genug Platz für die neue Klasse vorhanden ist. Eine Klasse K im Zwischenspeicher wird mit der Wahrscheinlichkeit A(K) ausgelagert. Die Umsetzung dieses Algorithmus ist einfach: Das Intervall [0, 1) wird unterteilt, jeder Klasse wird ein Teilintervall zugeordnet, dessen Länge A(K) beträgt. Anschließend wird eine Zufallszahl aus dem Intervall [0, 1) ausgewählt. Die Klasse, in dessen Teilintervall die Zufallszahl liegt, wird ausgelagert.

Bei der Umsetzung kann auf das Normieren der Intervallänge auf 1 verzichtet werden, dadurch müssen die Längen der Teilintervall nicht neu berechnet werden nachdem eine Klasse ausgelagert wurde.

5.1.3 Vergleich der Bewertungsfunktionen

Die Auswahl der günstigeren Bewertungsfunktion kann nicht durch eine Simulation unterstützt werden. Entscheidend für das Ergebnis einer Simulation sind die Eingangsdaten. Wird in den Simulationsdaten die Grundlage des LRU (siehe Seite 63) berücksichtigt, indem häufig mehrere Zugriffe auf eine Klasse ausgeführt werden, dann wird der LRU voraussichtlich besser abschneiden. Geht man dagegen von völlig zufälligen Zugriffen auf Klassen aus, dann kann der LRU gegenüber dem Random-Algorithmus keine Vorteile ausweisen.

Es soll daher untersucht werden, welchen Einfluß wiederholte Zugriffe auf eine Klasse für den Erfolg der LRU besitzt. Für den Vergleich der beiden Verfahren ist die Berücksichtigung aller Einflußfaktoren nicht notwendig. Der Zwischenspeicher faßt exakt m Klassen und alle Klassen besitzen die gleiche Größe, eine Klasse ersetzt folglich genau eine andere Klasse. Die Übertragungszeit ist für alle Klassen gleich groß, dieser Aspekt wird später berücksichtigt.

Die Abbildung 14 zeigt wie die Wahrscheinlichkeit abnimmt, daß sich eine Klasse noch im Zwischenspeicher befindet, wenn die Zahl der Zugriffe auf neue Klassen zunimmt. Für den LRU-Algorithmus ist die Wahrscheinlichkeit, daß die Klasse noch im Zwischenspeicher ist, eins, wenn weniger als m Klassen ausgelagert wurden, danach ist die Wahrscheinlichkeit Null. Der LRU-Algorithmus lagert die Klasse aus, die am längsten nicht benutzt wurde. Wurden nach einem Zugriff m-1 mal neue Klasse gespeichert, so wird die Klasse als nächste ersetzt. Beim Random-Algorithmus nimmt die Wahrscheinlichkeit für das Verbleiben der Klasse dagegen stetig ab. Der Verlauf der Kurve ist von m abhängig, nähert sich jedoch für m gegen Unendlich der Funktion

$$\text{für } m \rightarrow \infty : W(x) = e^{-x} \quad (x \geq 0)$$

an (ohne Beweis).

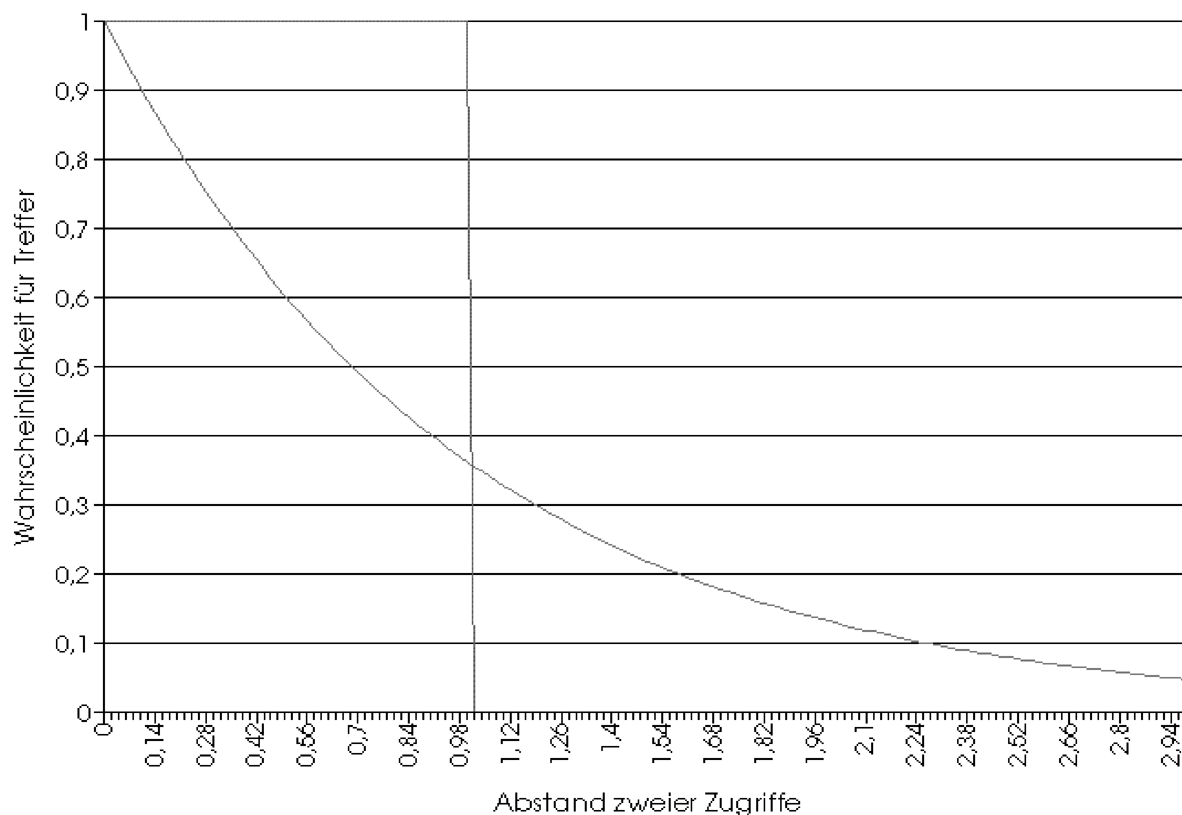

Abbildung 14: Wahrscheinlichkeit, daß die Klasse noch im Zwischenspeicher ist - in Abhängigkeit der zwischenzeitlichen Zugriffe auf andere Klassen.

Das praktische Verhalten der Bewertungsverfahren ist aus der Abbildung 14 nicht direkt abzuleiten. Interessant ist eine Betrachtung, die berücksichtigt, daß die Abstände zweier Zugriffe auf eine Klasse für jede Klasse unterschiedlich ist. Einigen Klassen werden voraussichtlich regelmäßig verlangt, während andere Klassen nur ein einziges mal benötigt werden. Es liegen noch keine Erkenntnisse über den Bedarf von Klassen im Agentensystem Mole vor. Der Bedarf an Klassen ist stark von der Agentenanwendung abhängig und unterliegt daher ständigen Veränderungen. Um auf eine Betrachtung nicht völlig verzichten zu müssen, wird an dieser Stelle eine Normalverteilung der Abstände zweier Zugriffe auf eine Klasse vorausgesetzt. Der Abstand gibt die Anzahl der zwischenzeitlich erfolgten Auslagerungen von Klassen an. Die Varianz der Verteilung wurde auf 1 festgelegt. Die Verteilung wird also durch folgende Funktion beschrieben:

$$V(x, \mu) = \frac{e^{-\frac{1}{2}(x-\mu)^2}}{\sqrt{2\pi}}$$

Die Normalverteilung stellt sich im Graph als »Glocke« dar, die symmetrisch zum Erwartungswert μ ist, die Verteilung ist für jeden x-Wert positiv. X ist hier der Abstand zweier

Zugriffe, da dieser allerdings nicht negativ werden kann, soll die Verteilung nur partiell für $x \geq 0$ definiert sein.

Wird die Verteilungsfunktion mit der obigen Wahrscheinlichkeit multipliziert und anschließend über das Intervall $[0, \infty)$ integriert, dann erhält man die Trefferquote des Zwischenspeichers, das heißt die Wahrscheinlichkeit, daß eine beliebige Klasse beim Zugriff noch im Zwischenspeicher ist. Da die Verteilung nur partiell definiert ist, ergibt die Fläche unter dem Graph der Verteilungsfunktion weniger als 1; die Trefferquote muß also entsprechend normalisiert werden. Die Tabelle 10 zeigt die Formeln der Trefferquoten für beide Bewertungsfunktionen.

Trefferquote bei LRU	**Trefferquote beim Random-Algorithmus**
$T(\mu) = \frac{\int_0^1 e^{-\frac{1}{2}(x-\mu)^2}\, dx}{\int_0^\infty e^{-\frac{1}{2}(x-\mu)^2}\, dx}$	$T(\mu) = \frac{\int_0^\infty e^{-\frac{1}{2}(x-\mu)^2 - x}\, dx}{\int_0^\infty e^{-\frac{1}{2}(x-\mu)^2}\, dx}$

Tabelle 10: Trefferquote der beiden Algorithmen in Abhängigkeit des Erwartungswerts der Normalverteilung für den Abstand der Zugriffe.

Durch die Wahl der Normalverteilung für die Abstände der Zugriffe, bezieht sich das Ergebnis nicht mehr auf eine bestimmte Klasse, sondern gilt für den gesamten Zwischenspeicher. Der Erwartungswert μ wurde hier nicht festgelegt wurde, deshalb ist es möglich die Trefferquote über dem Erwartungswert für die Abstände aufzutragen.

Die Verläufe der Graphen in Abbildung 15 bestätigen das erwartete Verhalten. Je mehr Klassen - zwischen zwei Zugriffen auf eine Klasse - ausgelagert werden (hier als Abstand zwischen den Zugriffen bezeichnet), desto geringer ist die Wahrscheinlichkeit, daß die Klasse noch im Zwischenspeicher vorliegt. Sind die Abstände aller Zugriffe normalverteilt mit dem Erwartungswert μ, dann ist der LRU für kleine Erwartungswerte günstiger als der Random-Algorithmus. Wird der Erwartungswert der Abstände größer, dann nähern sich die Trefferquoten an, bis schließlich der Random-Algorithmus besser abschneidet.

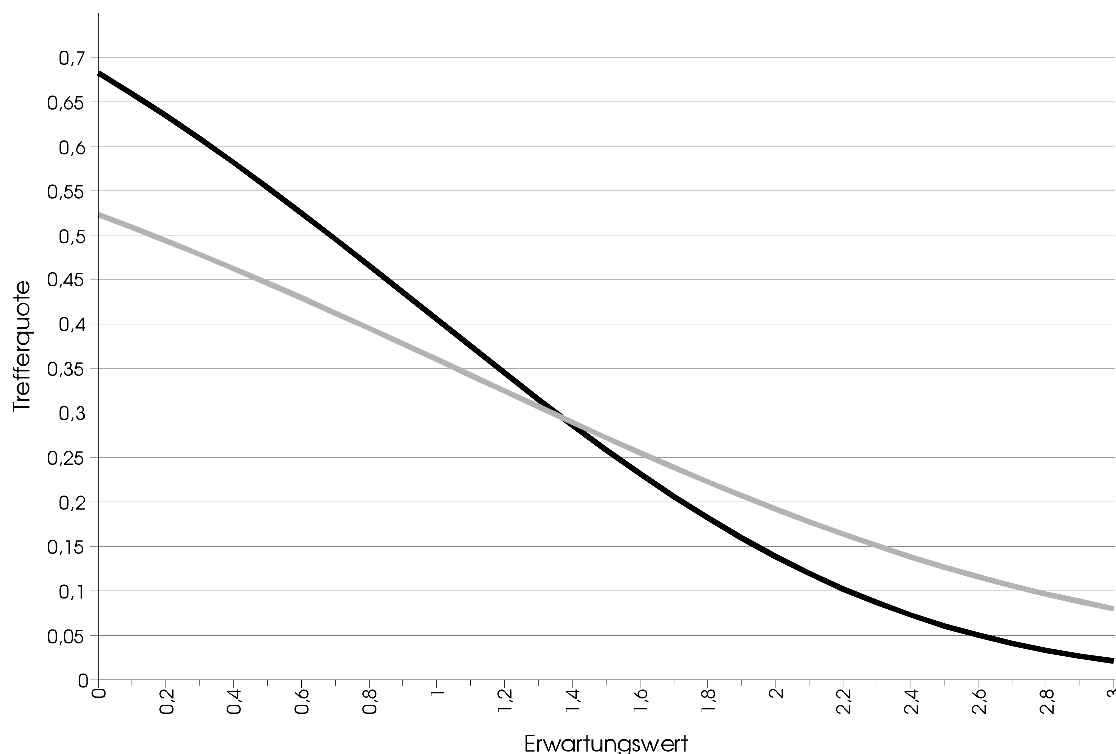

Abbildung 15: *Die LRU-Trefferquote (= Cache-Hitrate = Wahrscheinlichkeit, daß eine Anfrage aus dem Zwischenspeicher bedient werden kann) in Abhängigkeit des Erwartungswertes der Verteilungsfunktion (am häufigsten auftretender zeitlicher Absand zweier Zugriffe auf eine Klasse) ist schwarz, die Trefferquote des random-Algorithmus ist punktiert dargestellt. Der Erwartungswert ist auf die Größe des Zwischenspeichers normiert.*

Letztlich kann nur ein Bewertungsverfahren implementiert werden. Für keines der beiden vorgestellten Verfahren kann ein besseres mittleres Verhalten festgestellt werden, solange die Charakteristik der Zugriffe nicht bekannt ist. Diese ist allerdings von der Agentenanwendung abhängig und kann sich jederzeit ändern. Wenn beide Verfahren in etwa gleichwertig sind, dann ist der Random-Algorithmus vorzuziehen, da dieser einfacher zu implementieren ist. Die Wahl des Bewertungsverfahrens kann zu einem späteren Zeitpunkt aufgrund gesammelter Kenntnisse erneut getroffen werden und fällt möglicherweise zugunsten einer anderen Bewertungsfunktion aus.

Die effiziente Auslastung des Zwischenspeichers stellt nur ein Möglichkeit dar, die Performanz des Serversystems zu verbessern. Nicht immer ist es möglich Anfragen aus dem Zwischenspeicher zu bedienen, in solchen Fällen müssen andere Server befragt werden. Eine weitere Verbesserung erreicht man, wenn zunächst Server befragt werden, die über leistungsfähige Verbindungen schnell erreichbar sind.

5.2 Schnelle Verbindungen suchen

Der Basismechanismus garantiert das Auffinden einer Klasse. Für die Performanz des Serversystems ist es entscheidend einen Server zu finden, der die Klasse anbietet und zu dem eine schnelle Verbindung besteht. Neben dem Heimatserver oder den Sekundärservern gibt es die noch die Möglichkeiten benachbarte Server zu befragen oder die Klasse vom Server des vorhergehenden Aufenthaltsorts eines Agenten zu besorgen.

5.2.1 Nachbarserver

Ein Server ist nicht aufgrund seines Standortes zu einem anderen Servern benachbart, vielmehr entscheidet die Qualität der Verbindung zu diesem Server, ob es sich um einen Nachbarn handelt. Ähnlich wie Menschen einander Nachbarschaftshilfe leisten, können sich auch Server gegenseitig unterstützen. Bevor eine Klasse über den Basismechanismus besorgt wird, kann ein Server seine Nachbarn nach der Klasse befragen. Wenn die Klasse dort vorhanden ist, dann entfällt eine teure globale Kommunikation für den Basismechanismus.

Die Zahl der Nachbarserver n wird für jeden Codeserver durch seinen Betreiber vorgegeben. Ein *Nachbarserver* ist ein Server zu dem eine Verbindung besteht, deren festgestellte Qualität nur von höchstens n - 1 anderen Serververbindungen überboten wird. Jeder Server sortiert die ihm bekannten Codeserver nach der gemessenen Qualität der Verbindungen zum jeweiligen Codeserver, die ersten n Codeserver dieser Rangordnung sind die Nachbarserver.

In der Definition eines Nachbarservers taucht der Begriff festgestellte Qualität auf. Wobei noch zu klären ist, was Qualität bedeutet und wie und wann sie festgestellt wird. Diesen Fragen wird ausführlich in der Diplomarbeit [Voigt 1996] nachgegangen. Hier gilt, daß die Qualität einer Verbindung hoch ist, wenn der Zeitbedarf eine Klasse zu übertragen gering ist. Die Qualität einer Verbindung kann also als Zeitdauer von der Anforderung bis zum eintreffen der Klasse gemessen werden. Andere Kriterien, wie Verbindungskosten oder ähnliches, bleiben hier unberücksichtigt. Der Server, der eine Klasse anfordert kann nur die Antwortzeit und die Übertragungszeit messen und damit die Gesamtzeit als Summe daraus berechnen (vgl. Abbildung 16).

Leider sind die gemessenen Zeiträume nur mit Vorsicht zu interpretieren, da die Kommunikation in Java bereits auf zahlreiche Mechanismen zurückgreift. Oft werden die Daten in Java erst ausgeliefert, wenn diese vollständig übertragen wurden, in diesem Fall kann nur die Gesamtzeit nach Abbildung 16 gemessen werden. Um die reine Antwortzeit zu ermittelt, sollten also keine Nutzdaten übertragen werden, aber auch diese Messungen sind nicht ohne Vorbehalt zu gebrauchen, wie die vorgenommenen Messungen (siehe Tabelle 3) zeigen. Die Antwortzeit kann nur durch Verwendung von UDP gemessen werden, allerdings entspricht diese nicht der Antwortzeit bei Verwendung von TCP (siehe Diskussion der Tabelle 3).

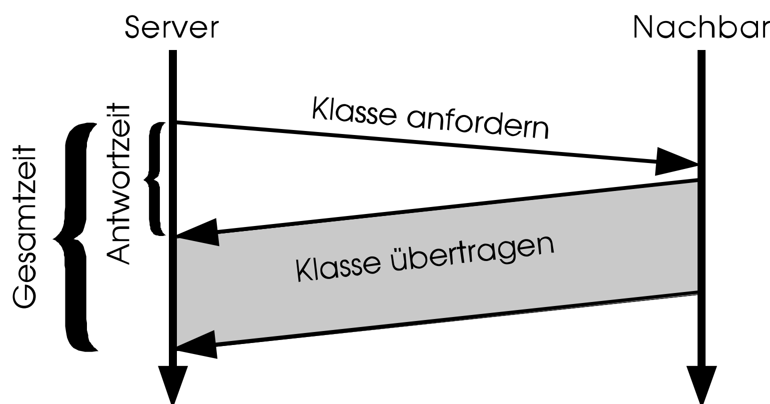

Abbildung 16: Zeiten bei der Übertragung einer Klasse

Die gemessenen Zeiten lassen Werte in der Größenordnung weniger Millisekunden erwarten. Sind die gemessenen Zeiten im Sekundenbereich, so handelt es sich sicherlich nicht mehr um einen benachbarten Server. Daher sollte es genügen, die Gesamtdauer für die Besorgung einer gewissen Datenmenge zu messen. Die Größe der Datenmenge sollte sich an der mittleren Größe einer Klasse orientieren (siehe Kapitel »Eigenschaften des Codes mobiler Agenten«). Es empfiehlt sich die gemessene »Entfernung« der letzten Messungen zu mitteln, damit Ausreißer nicht zu unnötigen Änderungen der Reihenfolge der Nachbarn führt.

Die ersten n Server mit der kleinsten Entfernung sollen bei einer Anfrage beim Codeserver nach der gewünschten Klasse befragt werden. Dies darf keinesfalls durch sequentielles Nachfragen bei diesen Servern geschehen, da dies viel zu lange dauern würde. Statt dessen sollte der Server jederzeit darüber informiert sein, welche Klassen seine Nachbarn speichert. Dazu führt der Server ein Register mit allen Klassen der Nachbarn. Dieses Register muß regelmäßig aktualisiert werden. Sollte im Register eine Klasse eingetragen sein, die der Nachbar nicht mehr bereitstellt, dann wird schlimmstenfalls eine unnötige Anfrage gestellt. Die Anforderungen an die Aktualität des Register sind folglich nicht sehr hoch. Dennoch müssen die Einträge regelmäßig aktualisiert werden, damit nicht zu viele Fehlzugriffe auftreten. Während die Fehlzugriffe für die Ausführung von Agenten im Agentensystem zusätzliche Verzögerungen darstellen, kann der Informationsaustausch zur Aktualisierung der Register asynchron erfolgen und eine Verzögerung der Ausführen eines Agenten wird vermieden.

Der Bestand an Klassen eines Servers kann unter Umständen umfangreich werden. Es ist daher nicht zweckmäßig ständig eine lange Liste aller Klassennamen auszutauschen. Eine Verlängerung der Zeitabstände des Informationsaustausch führt dagegen zu zahlreichen Fehlzugriffen beziehungsweise zu vielen Aufrufen des Basismechanisums, da die Klasse bei den Nachbarn scheinbar nicht vorhanden war. Als Alternative kann der Server entscheiden, ob er eine komplette Liste aller Klassen haben möchte, oder ob er nur die Änderungen seit dem letzten Austausch bekommen möchte. Wird die Liste der Änderungen länger als die

gesamte Klassenliste, dann kann der Nachbarserver statt den Änderungen einfach die gesamte Liste senden.

Auch wenn bislang durch die Formulierung »Austausch von Informationen« impliziert wurde, daß die Nachbarschaftsbeziehung symmetrisch ist, so ist dies nicht der Fall. Es kann durchaus vorkommen, daß die Zahl der Nachbarn eines Servers auf Null festgelegt wird. In diesem Fall kann der Server zwar als Nachbar agieren, aber besitzt selbst keine Nachbarn. Außerdem kann ein Server der Nachbar beliebig vieler anderer Server sein. Der Server sollte die Liste der Änderungen zwar für jeden Server dessen Nachbarn er ist aufbereiten, aber er sollte nicht jeweils eine Kopie anlegen, da dies Verwendung von Ressourcen ist. Damit die Liste der Änderungen nicht unbeschränkt wachsen kann, muß der Server die Liste regelmäßig kürzen.

Der Server erstellt ein Protokoll, in dem er jede neue Klasse mit eine Pluszeichen »+« und jede gelöschte Klasse mit einem Minuszeichen »-« versieht. Bei jeder Abfrage der gespeicherten Klassen beziehungsweise der Änderungen wird ein Merkpunkt in das Protokoll eingefügt, jeder dieser Merkpunkt besitzt eine innerhalb des Servers eindeutige Nummer. Die Angabe einer Merkpunktnummer teilt dem Server bei einer Anfrage nach den Änderungen mit, welches der letzte bekannte Stand war. Der Server kann dann alle Protokolleinträge, die nach diesem Merkpunkt stehen, übermitteln. Anschließend wird ein neuer Merkpunkt an das Protokoll angefügt und der alte Merkpunkt gelöscht. Es müssen nur Protokolleinträge gespeichert werden, die nach einem Merkpunkt stehen.. Gehen einem Protokolleintrag keine Merkpunkte voraus, dann kann dieser Eintrag gelöscht werden. Wächst das Protokoll aufgrund eines alten Merkpunktes über eine bestimmte Grenze, so wird dieser Merkpunkt einfach gelöscht. Anfragen, die sich auf nicht existente Merkpunkte beziehen, werden durch die Liste aller gespeicherten Klassen beantwortet. Sollte ein benachbarter Server ausfallen, dann wird er sich nie wieder auf seinen Merkpunkt beziehen, durch die obige Alterungsregel für Merkpunkte wird dieser überflüssige Merkpunkt bei Bedarf einfach aufgeräumt. Analog zur Mathematik heben sich gleiche Terme mit unterschiedlichem Vorzeichen auf, das heißt das Löschen und erneute Laden (oder das Laden und Löschen) einer Klasse kann aus dem Protokoll gelöscht werden, sofern sich keine Merkpunkte zwischen diesen Protokolleinträgen befindet. Im Anhang befinden sind Beispielprotokolle, welche die Funktionsweise verdeutlichen sollen.

Der beschriebene Protokollmechanismus kommt mit konstantem Speicherbedarf aus, der Speicherplatz kann als Ringpuffer implementiert werden. Wird ein Merkpunkt überschrieben, dann gilt er als veraltet.

Aufgrund der Liste der Änderungen kann jeder Server sein Register der beim Nachbarn gespeicherten Klassen ständig aktualisieren. Bei der Suche nach einer Klasse können die Klassenbestände der Nachbarn bereits lokal durchsucht werden.

Das Intervall, in dem die Änderungen abgefragt werden, sollte nicht festgelegt werden. Wird das Intervall zu kurz gewählt, dann haben sich seit der letzten Abfrage möglicherweise keine Änderungen ergeben, der Nachrichtenaustausch wäre überflüssig. Ist das Intervall zu

groß, dann ergeben sich viele Fehlabfragen oder bereits vorhandene Klassen werden nicht berücksichtigt. Eine dynamische Anpassung der Intervallänge an die aktuelle Situation ist also sinnvoll. Die Intervallänge kann durch folgende Regeln gesteuert werden:

- Je weniger Änderungen sich im letzten Zeitintervall ergaben, desto stärker muß das Intervall verlängert werden.
- War die Anzahl der Änderungen groß, dann sollte die Intervallänge verkürzt werden.
- Bei einer Fehlabfrage muß die Intervallänge deutlich verkürzt werden.

Da diese Einflußgrößen für jeden Nachbarn unterschiedlich sind, sollte die Intervallänge für jeden Nachbarn getrennt ermittelt werden. Der Erfolg des Regelprozesses hängt entscheidend von der Gewichtung aller berücksichtigten Einflußgrößen ab. Folgende Formel stellt einen Vorschlag für die Gewichtung der Einflußgroßen dar: $I_n = I_{n-1}\left(1 + p\left(A_{n-1} - A_{soll}\right)\right) 2^{-F}$

I ist die Intervallänge, A ist die Anzahl der Änderungen pro Intervall, p dient zur Gewichtung und liegt in der Größenordnung von Prozent (0,002..0,05), F gibt die Anzahl der Fehlabfragen an. Mit dem Parameter p kann eingestellt werden zu welchem Anteil die Differenz zwischen der Anzahl der Änderungen und der gewünschten Anzahl an Änderungen berücksichtigt wird. Ist p = 1%, dann wird pro Abweichung von der gewünschten Anzahl Änderungen das Intervall um 1% gekürzt bzw. verlängert. Ist p sehr klein, so ändert sich die Intervallänge nur langsam, bei großem p neigt die Folge zu starken Schwankungen. Bei jedem Fehlzugriff wird die Intervallänge halbiert.

Es sollen nur die Klassen der Nachbarn im Register gehalten werden. Wird ein Server zum Nachbarn, so sind alle Klassen dieses Servers abzufragen, umgekehrt kann ein Server die Nachbarschaft verlassen, dann müssen dessen Klassen aus dem Register entfernt werden. Dies kann zu häufigen und umfangreichen Änderungen des Registers führen, wenn mehrere Server um die letzten Plätze der Nachbarschaft konkurrieren. Um häufige und umfangreiche Änderungen des Registers zu vermeiden wird ein Hysterese-Bereich eingeführt. Das Prinzip soll anhand eines Beispiels verdeutlicht werden.

Ein Server besitzt 10 Nachbarn und einen Hysterese-Bereich von 2 Servern (siehe Abbildung 17). Verdrängt ein neuer Nachbar den letzten Nachbarn (Nachbar 10), dann gerät dieser in den Hysterese-Bereich (Hysterese 1) und wird als normaler Nachbar weitergeführt. Der Server Hysterese 1 wandert auf den zweiten Hysterese-Platz und der Server, der dort war wird aus dem Register entfernt. Wichtig ist, daß neue Server nicht auf den Hysterese-Plätzen einsteigen können. Ein Server kann sich nur als Nachbar qualifizieren, wenn er einen regulären Nachbar-Platz (1 bis 10) erreicht. Dagegen kann ein Server auf einem Hysterese-Platz ohne Einschränkung wieder einen Nachbar-Platz einnehmen, die dazugehörigen Daten wurden ständig aktualisiert, so daß kein zusätzlicher Aufwand entsteht.

Bei geeigneter Wahl der Größe des Hysterese-Bereichs kann das ständige Wechseln der Server im Register verhindert werden. Die Wahl der Größe ist auch davon Abhängig, wie

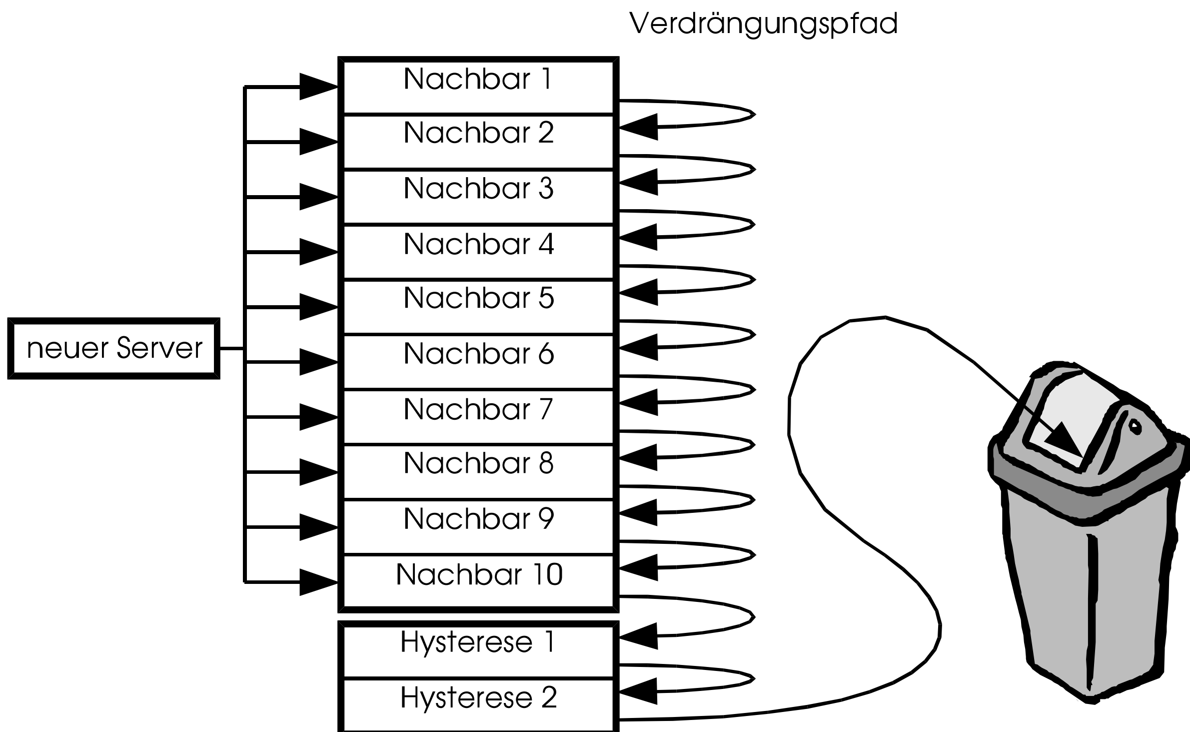

Abbildung 17: Führung des Nachbarschaftsregisters am Beispiel von 10 Nachbarn

stark aktuelle Meßergebnisse bei der Rangfolge der Nachbarn berücksichtigt werden (siehe oben).

Neben der Verwaltung der Nachbarserver und dem Informationsaustausch unter diesen, ist interessant woher ein Server von der Existenz seiner Nachbarn erfährt, für dieses Problem gibt es eine Vielzahl möglicher Lösungen. Das Problem taucht in ähnlicher Form auch an anderer Stelle auf, zum Beispiel müssen die Agenten in Erfahrung bringen welche Ort es gibt. Die hier vorgestellten Lösungen sind teilweise auch für die anderen Problemstellen realisierbar.

- Statische Konfiguration
  Der Server bekommt beim Start durch den Benutzer eine Menge möglicher Nachbarn mitgeteilt. Diese Möglichkeit eignet sich vor allem in der Anlaufphase eines Systems, der Nachteil der statischen Konfiguration liegt darin, daß sie sich nach dem Start des Servers nicht mehr ändert.
- Zentraler Auskunftsdienst
  An zentraler Stelle könnten alle Adressen der Server abgelegt werden. Allerdings ist die Auswertung dieser umfangreichen Adressensammlung schwierig. Das Aktualisieren des zentralen Datenbestandes ist mit hohem Aufwand verbunden, da der Ausfall eines Servers erkannt werden muß. Die zentrale Stelle der Adreßverwaltung macht das Gesamtsystem empfindlich gegenüber Netzpartionierung und anderen Fehlern und kann bei hoher Last zu einem Engpaß werden.

- Heimatadressen
  Durch die Anforderung von Klassen wird dem Server die Heimatadresse der Klasse mitgeteilt. Bei dem Heimatserver kann es sich um einen potentiellen Nachbarn handeln, die Adresse sollte auf eine mögliche Nachbarschaft hin überprüft werden. Analog gilt die Überlegung für die Adressen von Sekundärservern in den Zusatzinformationen einer Klasse.
- Absenderrecherche
  Die Absenderrecherche entspricht der Umkehrung der Auswertung von Heimatadressen. Wendet sich ein Server an den Heimatserver, so wird dem Heimatserver automatisch der Server bekannt, der sich an der Heimatserver gewendet hat. Dies gilt nicht nur für die Rückverfolgung von Nachfragen beim Heimatserver, sondern analog auch für die andern Möglichkeiten.
- Besorgungshinweise
  Das Agentensystem teilt dem Server eine Adresse mit (siehe unten).

Bis auf den zentralen Auskunftsdienst sind alle Möglichkeiten gut miteinander zu kombinieren. Ohne den zentralen Auskunftsdienst sind immer Fälle zu finden, in denen zwei unmittelbar nebeneinanderstehende Server sich nie gegenseitig kennenlernen, weil sie weder in der statischen Konfiguration auftauchen, noch Klassen beheimaten und auch keine Agenten direkt zwischen ihnen migrieren. Die Wahrscheinlichkeit für das Eintreten aller dieser Fälle dürfte gering sein. Tritt ein solcher Fall dennoch ein, dann gibt es zwei Nachbarn, die sich nicht kennengelernt haben, das Laufzeitverhalten ist dadurch nicht optimal, jedoch ist das Funktionieren des System dadurch nicht beeinträchtigt.

5.2.2 Besorgungshinweise

Der Codeserver ist nicht in der Lage herauszufinden von wo ein Agent kam oder wohin ein Agent gehen möchte. Das Agentensystem kennt jedoch den letzen Aufenthaltsort des Agenten. Es ist wahrscheinlich, daß sich am vorhergehenden Aufenthaltsort noch Klassen befinden, die am neuen Ort wieder benötigt werden. In der bestehenden Mole-Implementierung werden die Klassen immer vom letzten Aufenthaltsort besorgt. Dies führt zwar zu Einschränkungen (siehe Kapitel »Bestätigtes Propagieren«), kann aber im Einzelfall schneller sein, als die Anwendung des Basismechanismus.

Bei einer Anfrage beim Codeserver kann das Agentensystem die Adresse des Codeservers übermitteln, der dem letzen Aufenthaltsort die Klassen bereitstellte. Diesen Hinweis kann der Codeserver verwenden um benötigte Klassen von dort zu besorgen, sofern die Klassen nicht schon lokal vorliegen oder durch einen Nachbar bereitgestellt werden können.

Neben dem Zeitvorsprung durch den Besorgungshinweis lernt der Codeserver möglicherweise einen Codeserver kennen, der benachbart ist. Noch schneller als das Ausnützen

schneller Verbindungen, ist der Zugriff auf lokale Dateien. Ideal wäre es, wenn Klassen schon besorgt würden, bevor das Agentensystem Bedarf danach hat.

5.3 Vorratshaltung

Ein Anfrage kann am schnellsten beantwortet werden, wenn die gefragte Klasse bereits im Codeserver vorgehalten wird. Eine Vorratshaltung von Klassen, die mit hoher Wahrscheinlichkeit in Kürze benötigt werden, ist sinnvoll, wenn sie möglich ist.

Welche Klasse im weiteren Verlauf eines Programms benötigt wird, ist nicht vorherzusehen. Es ist jedoch möglich aus den in der Vergangenheit benötigten Klassen auf den zukünftigen Bedarf zu schließen. Hat eine bestimmte Klasse eine andere schon mehrmals referenziert, dann ist es wahrscheinlich, daß die Klasse wieder referenziert wird.

Ein Beispiel soll verdeutlichen welche Informationen gesammelt werden können und wie diese Informationen verwendet werden können. Die Klasse K referenziert die Klassen A, B, C und D. Bei 437 Zugriffen auf K wurden anschließend die Klassen A-D wie folgt referenziert:

	A	B	C	D
Anzahl der Referenzen von Klasse K aus	26	437	437	381

Tabelle 11: Anzahl Referenzierungen der Klassen A, B, C und D bei 437 maliger Verwendung von Klassen K

Die Statistik sagt aus, daß die Klassen B und C immer referenziert wurden, wenn die Klasse K benötigt wurde. Es ist also wahrscheinlich, daß bei einer Anfrage der Klasse K in Kürze auch die Klassen B und C benötigt werden. Die Klasse D wurde in 87%, A in nur knapp 6% aller Fälle benötigt. Daraus läßt sich vermuten, daß es sinnvoll ist, D vorsorglich zu besorgen, während A voraussichtlich nicht benötigt wird.

Die Statistik kann nicht den Anspruch erheben, daß die Klasse auch zukünftig Klassen in gleicher Häufigkeit referenziert. Das stellt jedoch kein Problem dar, schließlich geht es nur darum, welche Klassen auf Vorrat besorgt werden sollen. Konnte der Bedarf einer Klasse nicht rechtzeitig vorhergesehen werden, dann wird diese erst auf Anforderung besorgt.

Ein Codeserver kann bei der Anforderung einer Klasse auch gleich die Klassen anfordern, die er mit hoher Wahrscheinlichkeit in Kürze benötigt. Je nach verfügbarer Kapazität gibt der Codeserver der Klassenanfrage eine Quote aus [0,1] mit. Der antwortende Server liefert

daraufhin alle Klassen zurück, deren Wahrscheinlichkeit der Nachfrage die Quote überschreitet. Liegt im obigen Beispiel die Quote bei 0,7, dann werden mit der Klasse K auch die Klassen B,C und D ausgeliefert, bei einer Quote von 0,9 wird nur noch B und C mitgeliefert.

Die Statistik, welche Klasse in Verbindung mit einer Klasse angefordert wurde, wird für jede Urklasse geführt. Eine Klasse wird zur *Urklasse*, wenn ein Agentenobjekt von dieser Klasse erzeugt wurde. Die Klassen der Objekte, die ein Agent mitführt, und die Superklassen des Agenten sind keine Urklassen. Die *globale Statistik* wird für alle Objekte der Urklasse gemeinsam geführt. Die *lokale Statistik* wird für alle Objekte im Bereich eines Codeservers gemeinsam geführt. Die Summe aller lokalen Statistiken einer Urklasse ergibt die globale Statistik. Die Statistik besteht aus einem Zähler für die Urklasse und jeweils einem Zähler für jede von ihr referenzierte Klasse.

Der Heimatserver führt die globale Statistik der Klasse. Alle anderen Server, die die Klasse speichern, können eine lokale Statistik führen. Der Stand der lokalen Zähler wird regelmäßig an den Heimatserver übertragen. Dieser addiert die lokalen Zählerstände auf die globalen Zähler und liefert die neuen globalen Zählerstände zurück. Die lokalen Zähler des Servers werden gelöscht und die aktuellen globalen Zählerstände werden zusätzlich gespeichert. Nur der Heimatserver darf den globalen Zählerstand ändern, alle anderen Server ändern nur ihre lokalen Zähler. Relevant für die Berechnung der Quote ist immer die Summe beider Zähler.

Der Heimatserver führt eine Alterung der globalen Zählerstände durch, indem er die eingehenden Meldungen über referenzierte Klassen stärker gewichtet, als die über längere Zeiträume gespeicherte Statistik. Durch eine Alterung kann berücksichtigt werden, daß sich die Verwendung einer schon lange bekannten Klasse ändert.

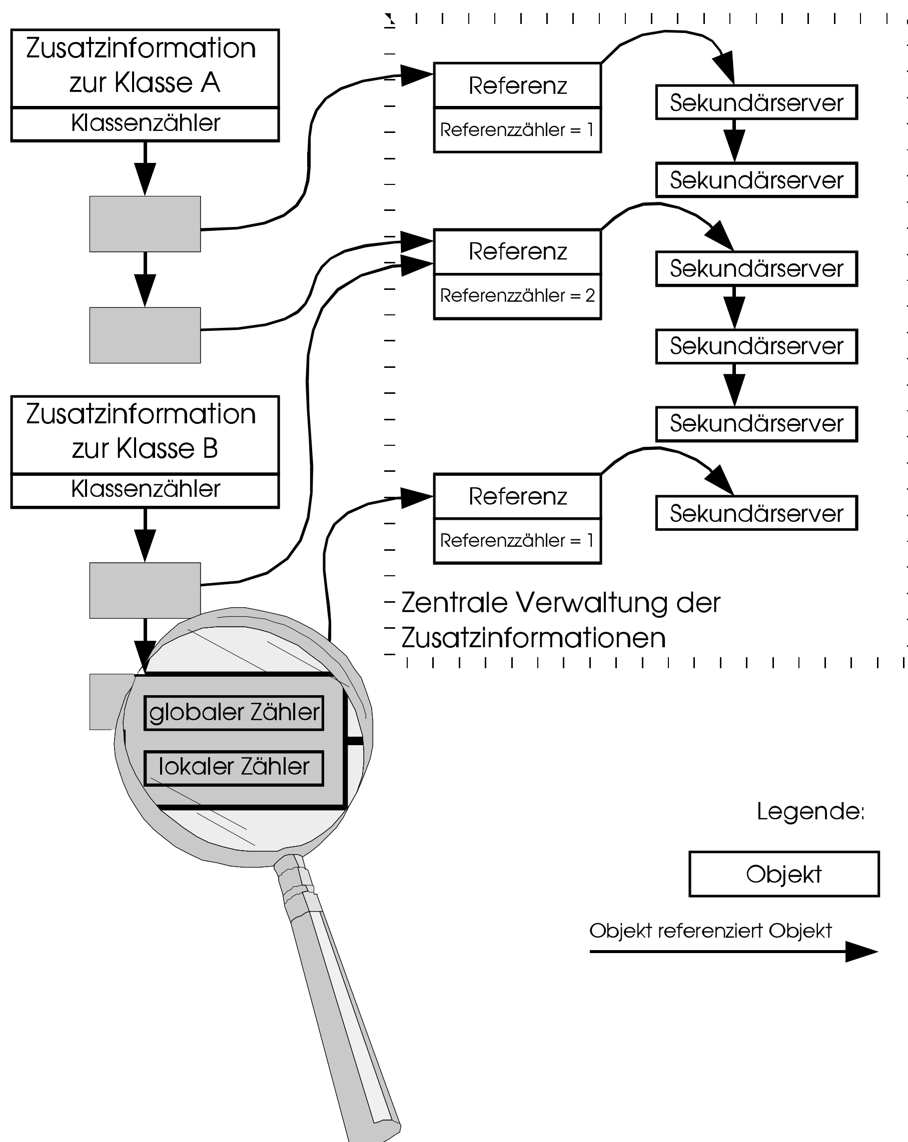

Abbildung 18: Kombinierte Verwaltung der Sekundärserver und der Zugriffsstatistik im Hauptspeicher des Servers

Das Wissen über die referenzierten Klassen und die Sekundärserver einer Klasse sind Zusatzinformationen. Es bietet sich folglich an, die Informationen über Sekundärserver und referenzierte Klassen gemeinsam zu halten und weiterzugeben. Neben den festgelegten Referenzen können durch die Statistik auch Laufzeitreferenzen berücksichtigt werden.

In der Abbildung 18 ist die Verwaltung der Zusatzinformation im Hauptspeicher des Servers dargestellt. Während die Sekundärserver einer Klasse im Codeserver für jede Klasse nur einfach gehalten werden, gilt dies für die Zugriffsstatistik nicht. Die lokale und globale Statistik wird für jede Referenzierung der Klasse getrennt geführt. Wurde eine Klasse n mal referenziert, dann wird die Zugriffstatistik n mal geführt, während die Sekundärserver nur einmal gespeichert werden.

Beim Sammeln der Informationen ist der Klassenlader im Agentensystem behilflich. Bei der Ankunft eines neuen Agenten wird im Agentensystem ein neuer Klassenlader erzeugt, der die Agentenklasse (die Urklasse) laden soll. Benötigt die Urklasse weitere Klassen, dann wird automatisch der zugehörige Klassenlader aktiviert. Der Klassenlader sollte sich also seine Urklasse merken, das ist die Klasse, die er als erste nach seiner Erzeugung geladen hat. Gegenüber dem Codeserver teilt der Klassenlader diese Urklasse bei jeder Anfrage mit, und ermöglicht so dem Codeserver eine Zuordnung der tatsächlichen Referenzen zu den Klassen.

5.4 Suchreihenfolge

Für die Suche nach einer Klasse stehen eine Reihe von Möglichkeiten zur Verfügung. Nicht alle Möglichkeiten liefern die Klasse gleich schnell. Bestimmte Möglichkeiten der Klassenbesorgung sind folglich anderen vorzuziehen.

Am schnellsten kann ein Codeserver eine Anfrage bedienen, wenn dem Server die Klasse bereits lokal vorliegt. Dies betrifft sowohl die Klassen, die der Basismechanismus lokal hält, wie auch die Klasse, die sich im Zwischenspeicher befinden. Wurde eine Klasse bereits auf Vorrat in den Zwischenspeicher gespeichert, dann kann die Klasse bei der Anfrage unverzüglich ausgeliefert werden.

Erst wenn die lokalen Möglichkeiten das Ergebnis nicht liefern können, dann muß mit anderen Servern kommuniziert werden. Per Definition ist die Kommunikation mit den Nachbarservern am schnellsten, daher wird zunächst der Klassenbestand der Nachbarserver nach der gewünschten Klasse durchsucht. Ist die gesuchte Klasse bei einem Nachbarn verfügbar, dann kann sie von diesem schnell besorgt werden.

Wenn die Klasse auch den Nachbarservern unbekannt ist, dann gibt es prinzipiell noch zwei Möglichkeiten, wo die Klasse zu suchen ist:

1. Wenn ein Besorgungstip des Agentensystems vorliegt, dann kann dieser Server befragt werden.
2. Der Basismechanismus liefert die Adressen von Servern, welche die Klasse speichern.

Berücksichtigt man, daß der Basismechanismus sowohl Heimat- als auch Sekundärserver benennen kann, dann liegt eine Menge von Adressen von Servern vor. Nur der Heimatserver garantiert, daß er die Klasse speichert, wenn sie Teil der Codebasis ist. Für alle anderen Server mag das Vorfinden der Klasse mehr oder weniger wahrscheinlich sein. Mit einer zufälligen Auswahl aus diesen Servern kann die durchschnittliche Zugriffszeit zwar nicht gesenkt

werden, aber es wird eine bessere Verteilung und damit eine gleichmäßigere Auslastung der verschiedenen Server erreicht. Wurde dabei nicht der Heimatserver ausgewählt, dann muß dieser in einem nächsten Anlauf befragt werden, sofern dem Server die Klasse unbekannt war.

Eskalations stufe	**Zu durchsuchender Klassenbestand**
1	Lokaler Klassenbestand des Basismechanismus
2	Lokaler Klassenbestand des Zwischenspeichers
3	Klassenbestand der Nachbarserver
4	Entweder ein Server aus dem Besorgungstip des Agentensystems oder der Heimat- bzw. ein Sekundärserver.
5	Heimatserver

Tabelle 12: Suchreihenfolge nach einer Klasse

Die Tabelle 12 faßt alle vorhandenen Möglichkeiten der Klassensuche zusammen. Die Suche beginnt mit der Eskalationsstufe 1, konnte damit kein Ergebnis erzielt werden, dann erhöht sich die Eskalationsstufe solange bis das Ergebnis der Klassensuche feststeht. Die Suche endet mit der Befragung des Heimatservers, da dieser, wenn er erreichbar ist, ein Ergebnis (Auslieferung der Klasse oder Bestätigung der Nichtexistenz) garantiert.

Der Codeserver besorgt Klassen immer erst bei Bedarf. Allerdings werden, nach der Auslieferung einer Urklasse, asynchron die Klassen besorgt, die mit hoher Wahrscheinlichkeit in Kürze von der Urklasse referenziert werden.

6 Sicherheit

Das Löschen oder Manipulieren von Daten kann, je nach Anwendung, einen beträchtlichen Schaden anrichten. Vor diesem Hintergrund muß das Agentensystem vor unbefugten Manipulationen geschützt werden. Für die Sicherheit innerhalb des Agentensystem muß das Agentensystem sorgen, für die sichere Bereitstellung der benötigten Klassen muß der Codeserver sorgen.

6.1 Prinzip und Zielsetzung

Auch das sicherste System kann Fehler in Programmen nicht verhindern. Der Zweck eines Programms legt der Programmierer fest. Bei der Registrierung des Programms kann der Codeserver nicht entscheiden, ob das Programm tatsächlich das gewünschte Ergebnis produziert. Erzeugt ein Programm unerwünschte Ergebnisse, dann kann es sich um einen versehentlichen Fehler handeln oder um ein böswillig verändertes Programm - der Codeserver kann die beiden Fälle jedenfalls nicht unterscheiden. Daraus folgt, daß die Klasse, so wie sie beim Server registriert wurde, als Original betrachtet wird. Geht bei einem beliebigen Codeserver eine Anfrage nach einer Klasse ein, dann ist es das Ziel des Serversystems, das unveränderte Original auszuliefern.

Dem Agentensystem wird im Rahmen dieser Arbeit völlig vertraut. Gelingt es einem Angreifer einen Agenten so zu manipulieren, daß er unbemerkt statt der gewünschten Klasse eine veränderte Klasse anfordert, dann sind sämtliche Sicherheitsvorkehrungen des Codeservers nutzlos. Der Codeserver geht folglich davon aus, daß der Agent die Klasse benötigt, dessen Namen er angibt. Solange das Serversystem und das Agentensystem sicher sind, ist diese Voraussetzung gewährleistet.

Das Grundprinzip ist einfach, eine Klasse wird beim Registrieren signiert und im Agentensystem wird die Signatur der Klasse überprüft. Die Abbildung 19 zeigt die Stationen die eine Klasse durchläuft. Nach dem Signieren ist die Klasse vor unbemerkten Manipulationen geschützt. Die Codeserver und die Komponenten, die für die Kommukation zuständig sind,

können die Klasse nicht unbemerkt verändern. Die Signatur Klasse wird erst im Agentensystem überprüft.

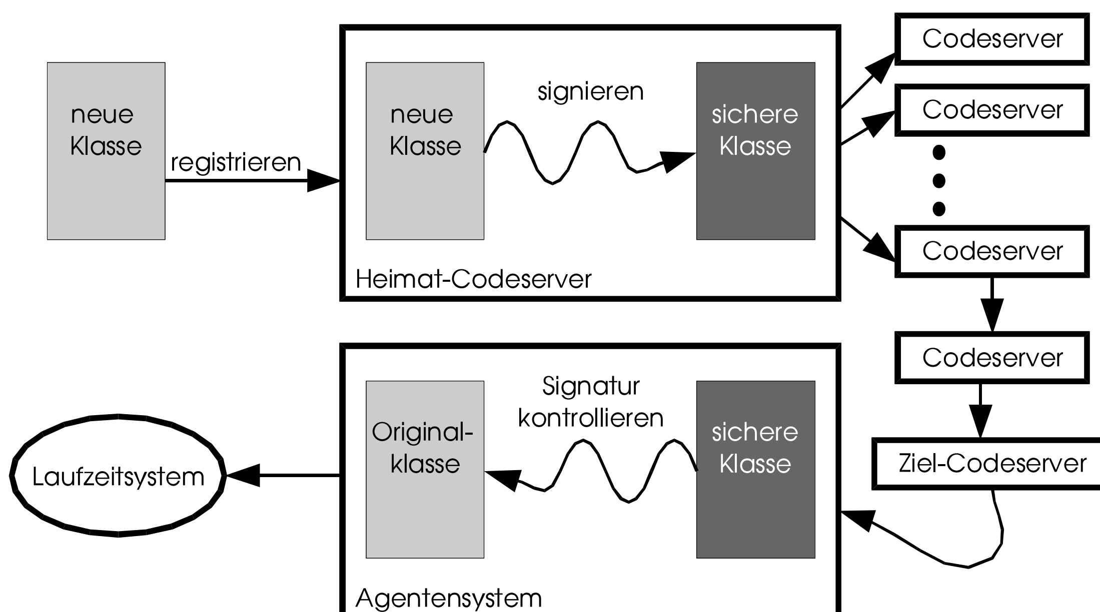

Abbildung 19: Prinzip der Sicherheitsmaßnahmen

Ein Angriff kann nur im Codeserver erfolgen, bei dem die Klasse registriert werden soll. Wurde dieser Codeserver manipuliert, dann kann die Klasse vor ihrer Registrierung manipuliert werden. Um diesen Angriff zu verhindern, müßte die Registrierung unter der Aufsicht des Benutzers durchgeführt werden. Allerdings kann der Benutzer die Registrierung nicht selbst durchführen und die Verwendung eines Hilfsmittels stellt eine weitere Angriffsmöglichkeit dar. Die Verantwortung für das unsichere Hilfsmittel trägt der ahnungslose Benutzer. Es ist daher sinnvoll, wenn die Registrierung durch den Codeserver vorgenommen wird. Durch Funktionskontrollen kann sich der Benutzer regelmäßig von der Sicherheit seines Codeservers überzeugen und Abweichungen dem Administrator melden.

Sicherheit ist keine Eigenschaft, die sich in ein beliebiges System integrieren läßt. Manipulationen am Betriebssystem, am Netzwerk oder am DNS führen zu Sicherheitslücken, die der Codeserver nicht verhindern kann.

Werden die Klassen sicher übermittelt, dann sind die Namen der von dieser Klasse referenzierten Klassen frei von Manipulationen, da das Class File Format die Namen der referenzierten Klassen einschließt. Dies gilt nicht für die Zusatzinformationen, da für diese keine Sicherheitsmaßnahmen vorgesehen sind. Die Zusatzinformationen enthalten nur die Adressen von Sekundärservern und die Statistik über die referenzierten Klassen. Manipulation dieser Informationen würde das Funktionieren des Agentensystems nicht gefährden. Existieren die Adresse einer manipulierten Sekundärserver-Angabe nicht, dann wird die entsprechende Angabe einfach ignoriert. Liefert der Codeserver einer fehlerhaften Sekundärserver-Angabe manipulierte Klassen zurück, wird dies bei der Sicherheitsüberprüfung (Kontrolle der Signatur) erkannt. Die statistischen Daten dienen ohnehin nur der Verbesserung der Per-

formanz und sind daher nicht sicherheitsrelevant. Auf das Signieren der Zusatzinformationen kann folglich verzichtet werden, es genügt wenn nur die Klasse signiert wird.

6.2 Sichere Schlüssel

Bei der Signatur kommen sogenannte Public-Key-Algorithmen zur Anwendung. Dafür benötigt man ein Schlüsselpaar, das besteht aus einem öffentlichen und einem privaten Schlüssel. Der private Schlüssel kann nicht aus der Kenntnis des öffentlichen Schlüssels gewonnen werden.

Das Hauptproblem beim Austausch von signierten Informationen besteht darin, daß der Empfänger den öffentlichen Schlüssel für die Nachricht benötigt. Dieser Schlüssel muß über einen sicheren Kanal ausgetauscht werden. Eine manipulierte Nachricht wird vom Empfänger nicht entdeckt, wenn der Empfänger den zur manipulierten Nachricht gehörenden Schlüssel verwendet.

Die Beschreibung und Implementierung eines Verfahrens zum sicheren Austausch von Schlüsseln würde den Rahmen dieser Arbeit sprengen. Ein Verfahren zum Erzeugen und sicheren Austauschen von Schlüsseln wird an dieser Stelle vorausgesetzt.

Die Java Cryptography Architecture [JavaJCA 1997] erweitert die Sprache Java um die Fähigkeit Verschlüsselungskonzepte einzusetzen und zu erweitern. Die Architektur basiert auf Anbietern von Sicherheitsmechanismen, einige bekannte Algorithmen sind bereits implementiert, andere können leicht hinzugefügt werden. Aufgrund der strengen Exportbestimmungen der USA bezüglich Verschlüsselungsverfahren darf die Java Cryptography Extension [JavaJCE 1997] außerhalb der USA nicht verwendet werden. Die Java Cryptography Extension enthält die Funktionen zum Verschlüsseln von Nachrichten und den Austausch von Schlüssel. Zum Umgang mit Sicherheitstechnologie wird bald eine internationale Regelung erwartet, danach ist mit einer internationalen Version der Cryptography Extension zu rechnen.

6.3 Sicherheit durch digitale Signaturen

Zum Signieren einer Klasse wird zunächst ein Schlüsselpaar erzeugt. Dann wird ein sicheres Abbildungsverfahren auf die Klasse angewendet, und das Ergebnis wird mit dem privaten Schlüssel verschlüsselt. Das verschlüsselte Ergebnis ist die sogenannte digitale Signatur der Klasse. Verfügt man über den öffentlichen Schlüssel, dann kann die Signatur entschlüsselt werden. Wird das sichere Abbildungsverfahren erneut auf die Klasse angewendet, dann muß das Ergebnis mit der entschlüsselten Signatur übereinstimmen.

Eine Manipulation der Klasse wird erkannt, da die Signatur der Klasse nicht mit der erwarteten Signatur übereinstimmt. Eine Manipulation der Signatur wird durch die Verschlüsselung verhindert. Das Raten einer passenden Signatur zu einer manipulierten Klasse, ist unwahrscheinlich und erfordert einen zu großen Rechenaufwand.

Die Java Cryptography Architecture [JavaJCA 1997] enthält bereits die Ansätze um digitale Signaturen zu erzeugen. Zum Einsatz kommen die Algorithmen DSA (Digital Signature Algorithm, beschrieben in NIST FIPS 186) oder RSA mit MD5 (Message Digest, definiert in RFC 1321). Der MD5 erzeugt eine 128 bit (16 Byte) lange Zahl mit folgenden Eigenschaften:

- Die Zahl ist schwer zu knacken, das heißt es sollte nicht möglich sein, eine Eingabe zu finden, zu welcher der MD5 die gleiche Zahl errechnet.
- Das Ergebnis des MD5 verrät nichts über die Eingabe, die zu dem Ergebnis führten.

Das Erzeugen der Schlüssel und das Signieren wird komfortabel unterstützt. Das Ergebnis der Signatur ist ein Array von Bytes, das die Klasse und die Signatur enthält. Der Codeserver beachtet den Inhalt des Arrays nicht, so daß die Signatur für den Codeserver transparent ist.

Das Agentensystem kann mit den Methoden der Java Cryptography Architecture das Array in einem Schritt überprüfen und die ursprüngliche Klasse zurückgewinnen. Durch die Überprüfung ist sichergestellt, daß die Klasse nicht manipuliert wurde.

Der Codeserver sollte nicht nur den Schutz vor Manipulationen gewährleisten, sondern auch Sicherheit beim Umgang mit verschiedenen Versionen einer Klasse.

6.4 Versionsverwaltung

Um die Konsistenz der Codebasis zu gewährleisten, wurde eine Änderung von registrierten Klassen ausgeschlossen. Nach der Änderung muß die Klassen einen anderen Namen tragen. Dies könnte dem Programmierer überantwortet werden, jedoch wird dieser dabei leicht überfordert, vor allem wenn umfangreiche Agenten geschrieben werden und zahlreiche Versionen ausprobiert werden sollen.

Der Name der Klassen wird nicht nur um die Adresse des Heimatservers erweitert (siehe Basismechanismus), es wird zusätzlich eine Versionsnummer für jede Klasse eingefügt.

Für Klassen, die der Benutzer in seinen Programmen verwendet, gelten folgende Vereinbarungen:

- Der Benutzer kann in seinen Programmen globale Klassennamen verwenden. Er bezieht sich dann auf genau die Klasse, die unter diesem globalen Namen registriert wurde.
  Beispiel: cembalo.dinformatik.duni-stuttgart.dde.p1234.v12.myPackage.MyClass
- Der Benutzer kann seine eigenen Klassennamen verwenden.
  Beispiel: myPackage.MyClass

Verwendet der Benutzer seine eigenen Klassennamen, dann sucht der Codeserver bei der Registrierung nach der neuesten Version dieser Klasse in seinem lokalen Bestand der Codebasis. Der Benutzer kann durch diese Vereinbarungen in seiner Arbeitsumgebung mit gewöhnlichen Klassennamen arbeiten. Der Codeserver nimmt die Abbildung auf globale Klassennamen bei jeder Registrierung erneut vor.

Der Programmierer übergibt dem Codeserver bei der Registrierung eine Menge von Klassen. Der Codeserver kontrolliert alle enthaltenen festgelegten Klassenreferenzen, diese müssen auf Klassen verweisen, die globale oder lokale Namen haben und vorhanden sind, oder sie verweisen auf Klassen aus der zu registrierenden Menge. Kann eine Referenz nicht aufgelöst werden, so ist die (überprüfbare) referenzielle Integrität der Codebasis (siehe Kapitel »Eigenschaften der Codebasis«) nicht gewährleistet, in diesem Fall wird keine Klasse registriert.

Klassen werden immer unter ihrem lokalem Namen dem Codeserver übergeben, dieser generiert einen globalen Namen, wobei er die Versionsnummer automatisch vergibt.

7 Schnittstellenbeschreibung

Die verschiedenen Schnittstellen wurden bereits in Kapitel 2.1.3 aufgeführt. Die Daten, die über diese Schnittstellen ausgetauscht werden sollen, wurden durch die vorangegangenen Kapitel beschrieben. In diesem Kapitel wird die Funktionalität der Schnittstellen zusammengefaßt.

7.1 Schnittstelle zum Programmierer

Ein Benutzer schreibt eine neue Klasse, die er in die Codebasis aufnehmen will. In der Regel greift der Benutzer auf bereits vorhandene Klassen zurück. Der Java-Compiler akzeptiert alle Verweise auf Klassen, die er in Pfaden finden kann, die in der Umgebungsvariablen CLASSPATH angegeben sind. Die benutzten Klassen bilden folgende Gruppen: Standardklassen, eigene Klassen, registrierte Klassen und sonstige Klassen.

Standardklassen sind die Klassen, die zum Sprachumfang von Java gehören. Diese werden nicht in die Codebasis aufgenommen, sondern müssen lokal verfügbar sein. Da diese Klassen Teil des Java-Systems sind, ist es erforderlich, daß die Klassen im Dateisystem abgelegt sind. Auch der Java-Interpreter selbst (die Java Virtual Maschine) muß selbstverständlich lokal vorliegen. Alle diese Teile sind abhängig von der ausführenden Maschine. Es gilt daher die Vereinbarung, daß alle Klassen, die sich in Paketen mit dem Präfix »java.« befinden, lokal vorhanden sein müssen. Um Manipulationen zu entgegnen, dürfen keine Standardklassen in die Codebasis aufgenommen werden. Auch die Klassen des Agentensystems sollten Standardklassen sein, die nicht durch den Codeserver verwaltet werden. Um nicht weitere Ausnahme-Pakete zu definieren, sollte zu jedem Codeserver besser eine Datei angelegt werden, in der die Namen der Standardklassen oder Standard-Pakete enthalten sind. Dadurch können kritische Klassen noch zu einem späteren Zeitpunkt durch den Betreiber des Codeservers zu Standardklassen erklärt werden. Erhält ein Codeserver eine Klassenanfrage einer Standardklasse, so weist er den Klassenlader an, die Klasse aus dem Dateisystem zu laden. Liegt die Klasse dort nicht vor, ist das ein Fehler, der vom System nicht behoben

werden kann. Die Liste der Standardklassen eines Codeservers muß mit den installierten Klassen der angeschlossenen Agentensystem übereinstimmen, damit dieser Fehler nicht auftreten kann. Fordert ein Codeserver bei einem anderen Codeserver eine Klasse an, die dieser als Standardklasse definiert, so tritt der gleiche Absprachefehler auf. Keinesfalls dürfen jedoch Standardklassen automatisch ausgetauscht werden, da auf diesem Weg sonst alle Sicherheitsmechanismen unterwandert werden. Einzig die Einrichtung eines zentralen und bekannten Anbieters von Standardklassen könnte die Sicherheit sinnvoll gewährleisten. Der zentrale Anbieter würde alle Standardklassen auf seinem Server signiert zur Verfügung stellen. Der öffentliche Schlüssel, mit dem die Klassen geprüft werden können, wird durch zahlreiche Veröffentlichungen allgemein bekannt gegeben. Ein automatischer Mechanismus zum Austausch von Standardklassen, zum Beispiel neue Module des Agentensystems, wird an dieser Stelle nicht weiter verfolgt.

Die zweite Gruppe bilden die eigenen Klassen, die ein Benutzer selbst geschrieben hat. Dabei handelt es sich ausschließlich um Klassen, die bereits beim zuständigen Codeserver registriert sind, oder bei diesem registriert werden sollen.

Der Gruppe der registrierten Klassen gehören alle Klassen an, die bereits auf einem beliebigen Codeserver registriert wurden. Diese Klasse tragen einen globalen Namen und sind dadurch leicht zu erkennen.

Der Programmierer ist bei der Arbeit mit dem Codeserver auf die Verwendung eines Programms, hier als Werkzeugs bezeichnet, angewiesen. Mit dem Werkzeug sollte der Benutzer Klassen besorgen, registrieren, löschen oder möglicherweise überflüssige Klassen abfragen können. Das Werkzeug kann diese Dienste dem Programmierer automatisch nach der Compilierung, in einer Kommandozeile oder menügesteuert anbieten.

Das Werkzeug steht zwischen dem Serversystem und dem Programmierer. Die Schnittstelle des Werkzeugs zum Programmierer muß dem Programmierer entgegenkommen - die Schnittstelle zwischen dem Werkzeug und dem Serversystem kann unabhängig davon gestaltet werden. Das Werkzeug wendet sich als Client an den Codeserver und der Codeserver reagiert nur auf die Anfragen des Werkzeugs. Die Client-Server-Kommunikation wird in Java durch die sogenannte RMI (Remote Methode Invocation, siehe [JavaRMI 1997]) komfortabel unterstützt.

Sun bezeichnet mit RMI, was unter dem Namen RPC (Remote Procedure Call) schon in anderen System verwendet wird. Mit dem RMI ist es möglich, ein Objekt auf einem entfernten Rechner verfügbar zu machen, so können von dem entfernten Rechner die Methoden des Objekts aufgerufen werden. Das Objekt ist allerdings nur scheinbar lokal verfügbar, denn der Aufruf der Methode wird an den Server übertragen und die gewünschte Methode des Objekts, das lokal auf dem Server vorhanden ist, wird aufgerufen. Das Ergebnis der Methode wird zurück an den entfernten Rechner übertragen und dem dortigen Methodenaufruf ausgeliefert. Weder der Aufrufer der Methode noch das Objekt beim Server muß sich um die Kommunikation kümmern, diese Funktionalität wird durch das RMI transparent bereitgestellt.

Das Werkzeug reicht die Aktionen des Benutzer in geeigneter Form an den Codeserver weiter. Die beiden Schnittstellen des Werkzeugs gleichen sich folglich im Inhalt exakt, nur die Form der Schnittstellen unterscheiden sich. Nachfolgend werden die Funktionen und ihre Parameter beschrieben, wobei diese Beschreibung keine exakten Datentypen vorgeben soll:

Funktionsname	**Parameter**	**Zweck der Funktion**
GetClasses	in: Name der Klasse out: Menge von Klassen im Class File Format (CFF)	Die statisch referenziert Klassen werden bei der Compilierung ebenfalls benötigt und werden deshalb ebenfalls besorgt. Dies setzt sich transitiv fort.
Register-Classes	in: geordnete Menge von Klassen im CFF out: geordnete Menge von Klassennamen	Eine Menge von Klassen soll registriert werden. Die globalen Namen werden erzeugt und zurückgegeben, der Vorgang kann gelingen oder scheitern, es ist nicht möglich nur einen Teil der Klassen zu registrieren
DeleteClass	in: Name der Klasse	Eine Klasse kann explizit gelöscht werden, alle Klassen, die diese Klasse referenzieren können zur Laufzeit nicht weiter ausgeführt werden.
unusedClasses	out: Menge von Klassennamen	Gibt eine Vorschlagsliste zurück, welche Klassen gelöscht werden könnten.

Tabelle 13: Funktionalität der Schnittstelle Codeserver-Programmierer

Selbstverständlich kann jede Funktion eine Ausnahmebehandlung auslösen, wenn die Funktion nicht durchgeführt werden konnte. Dabei werden die Funktionen entweder vollständig ausgeführt oder der Aufruf hat keine Auswirkungen.

7.2 Schnittstelle zum Agentensystem

Jede Klassenanforderung, die der Klassenlader im Agentensystem erhält muß dieser an den Codeserver weiterleiten. Diese Schnittstelle sollte so gestaltet sein, daß sich der Codeserver entfernt oder lokal zum Agentensystem befinden kann. Auch hier bietet sich die Verwendung der RMI an. Da die Methodenaufrufe für die Programme transparent weitergeleitet werden, ist es mit geringem Aufwand möglich, die entfernten Methodenaufrufe in lokale Methodenaufrufe umzusetzen. Nach einer Compilierung steht eine neue Version des Agen-

tensystems mit integriertem Codeserver zur Verfügung. Damit wird den Überlegungen der Anbindung des Codeservers an Mole entsprochen (siehe Kapitel »Anbindung des Servers an Mole«).

Ein Nachteil besitzt die Verwendung des RMI allerdings. Das Agentensystem ist mit allen Komponenten noch in der Java-Version 1.0 geschrieben. Leider ist es bislang nicht gelungen, entfernte Methodenaufrufe zwischen den Version 1.1 und 1.0 durchzuführen. In der alten Version gehört die RMI allerdings auch noch nicht zum Sprachumfang. Nachdem die RMI in der neuen Version zum Sprachumfang gehört, kann erwartet werden, daß künftige Versionen zur aktuellen Version kompatibel bleiben. Eine Portierung des Agentensystem auf die Version 1.1 ist beabsichtigt, so daß sich dieses Problem durch Warten löst.

Die folgende Aufstellung gibt die Funktionalität der Schnittstelle wieder, die einzelnen Funktionen können direkt in Methoden umgesetzt werden, die Beschreibung der Parameterliste läßt allerdings noch Freiheitsgrade bei der Umsetzung.

Funktionsname	**Parameter**	**Zweck der Funktion**
Tip	in: Name der Urklasse, Adresse des Servers	Das Agentensystem teilt dem Codeserver mit, daß die angegebene Klasse (und deren Referenzen) mit hoher Wahrscheinlichkeit beim angegebenen Server vorliegen.
GetClasses	in: Name der Urklasse, Name der Klasse out: signierte Klasse im Class File Format (CFF)	Besorgt die angeforderte Klasse. Diese Klassenanforderung wird bei den statistischen Daten der Urklasse berücksichtigt.
GetAllClasses	in: Name der Urklasse out: Menge von signierten Klassen im CFF	Mit GetAllClasses teilt das Agentensystem mit, daß es die angegebene Klasse und alle von dieser Klasse referenzierten Klassen benötigt. Diese Möglichkeit ist Agentensystemen vorbehalten, die nicht permanent am Netzwerk angeschlossen sind (zum Beispiel Laptops)

Tabelle 14 : Funktionalität der Schnittstelle Codeserver-Agentensystem

Für mobile Computersysteme ist eine Sonderbehandlung vorgesehen, da diese System nicht permanent am Netzwerk angeschlossen sind. Nach dem Eintreffen des Agenten bei einem solchen Agentensystem sollen alle benötigte Klassen besorgt werden. Sind alle benötigten Klassen installiert, dann kann der Agent seine Aufgaben mit hoher Wahrscheinlichkeit erledigen während das System ohne Verbindung arbeitet. Beim nächsten Kontakt mit dem Netzwerk kann der Agent sofort zur nächsten Lokation migrieren. Eventuell kehrt der Agent auch nur in seine Heimat zurück und informiert seinen Benutzer über die gesammelten

Ergebnisse und beendet sich danach. Welche Klassen benötigt werden, kann nur aufgrund des Bedarfs an Klassen in der Geschichte aller Agenten der Urklasse beurteilt werden. Um sicherzustellen, daß keine Klasse fehlt, müssen alle jemals benötigten Klassen ausgeliefert werden.

Das vorausschauende Besorgen von Klassen ist für ein mobiles Computersystem erforderlich, bei permanent angeschlossenen Agentensystemen kann es zur Verbesserung der Performanz eingesetzt werden. Die Schnittstelle zwischen zwei Servern berücksichtigt diese Option durch Wahrscheinlichkeitsangaben.

7.3 Schnittstelle innerhalb des Serversystems

Erhält ein Server die Aufgabe, eine bestimmte Klasse zu besorgen, dann ist der Server unter Umständen auf die Hilfe anderer Server angewiesen. Bringt der Server in Erfahrung, welcher Codeserver in Besitz der Klasse ist, dann tritt er diesem gegenüber als Client auf und fordert die Klasse und deren Zusatzinformationen an. In Java wird das Client-Server-Paradigma durch die RMI (siehe oben) komfortabl unterstützt. Die Frage, wie komplexe Objekte ausgetauscht werden können, wird mit der Verwendung der RMI gelöst.

Die Schnittstelle sollte folgende Funktionen ermöglichen:

Funktionsname	Parameter	Zweck der Funktion
GetClass	in: Name der Klasse, Quote der mitzuliefernden Klassen out: geordnete Menge von Klassen im CFF, geordnete Menge von Zusatzinformationen	Ein Server benötigt die angegebene Klasse, die Quote gibt da, welche der referenzierten Klassen mitgeliefert werden sollen. Beträgt die Quote zum Beispiel ½, dann werden alle Klassen mitgeliefert, die in der Hälfte aller Anwendungsfälle benötigt wurden. Das Ergebnis ist die gewünschte Menge von Klassen und zu jeder Klasse die gesammelten Zusatzinformationen.

Funktionsname	Parameter	Zweck der Funktion
Information-Exchange	in: Name der Urklasse, lokale Zahl der Anwendungen, geordnete Menge von Klassennamen, geordnete Menge von lokalen Quoten out: Zahl der Anwendungen, globale Anzahl Anwendungen, geordnete Menge von Klassennamen, geordnete Menge von globale Quoten	Die lokale Zugriffsstatistik, die ein Server gesammelt hat, wird dem Heimatserver übermittelt. Für jeden einzelnen Agenten wird notiert, wie oft welche Klasse angefragt wurde. Die gleichen Informationen werden dem Server zurückgeschickt, allerdings beziehen sich die Daten dann auf die Zugriffe bei allen Agentensystemen. Durch die größere Basis der Statistik trifft, nach dem Gesetz der Großen Zahlen, die Quote (siehe oben) im Mittel genauer zu.
GetTime-Delay	in: Menge von beliebigen Daten out: beliebige Daten	Der Initiator dieser Funktion mißt die Zeitdauer, bis das Ergebnis der Funktion vollständig ist. Diese Zeitdauer beinhaltet die Netzwerkverzögerung, die Übertragungsdauer und die Bearbeitungsdauer durch den angesprochenen Server jeweils unter der momentanen Last.
YourContents	in: letzter Merkpunkt bzw. null out: neuer Merkpunkt, Menge von Änderungsoperationen (+/- Klassennamen)	Ein anderer Server möchte sich über die Klassen informieren, die dieser Codeserver speichert. Der Merkpunkt zeigt den letzen bekannten Stand an. Wurde noch keine Inhaltsangabe erfragt, dann wird statt dem letzten Merkpunkt »null« übergeben und es werden alle Klassen zurückgegeben.

Tabelle 15 : Funktionalität der internen Serverschnittstelle

Mit der Erweiterung des Codeservers könnten weitere Funktionen hinzukommen. Diese Erweiterungen der Schnittstelle sind jedoch unproblematisch, da sie nur die Codeserver betreffen, externe Komponenten sind von Änderungen dieser Schnittstelle nicht betroffen.

8 Zusammenfassung und Ausblick

Die Zustandsdaten des Agenten stellen dessen »Persönlichkeit« dar, der benötigte Klassencode ist dagegen für eine ganze Reihe von Agenten charakteristisch. Bei der Migration ist die Trennung der Migration der Zustandsdaten von der Codemigration sinnvoll, da dadurch Laufzeitvorteile errungen werden können. Liegt der Code auf mehreren Servern vor, so kann derjenige ausgewählt werden, zum dem die schnellste Netzwerkverbindung besteht.

Bei der Betrachtung von einigen hundert Klassen konnten die wichtigsten Eigenschaften der Klassen in ihrer Größenordnungen ermittelt werden. Über die Verhaltensweisen der Agenten liegen dagegen keine Angaben vor, Verhaltensmuster von Agenten konnten durch Betrachtungen am Agentensystem nicht gewonnen werden, da dies der Stand des Projekts noch nicht zuließ. Der Codeserver sollte folglich offen für neue Verwendungsideen der Agenten sein, dies erfordert eine möglichst maximale Flexibilität bei der Bereitstellung der Agentenklassen.

Primäres Ziel des Codeservers ist es, dem Agentensystem eine Klasse zu besorgen, von welcher dem Codeserver lediglich der Name bekanntgegeben wird. Die Randbedingungen (siehe Kapitel »Restriktionen«) schließen eine zentrale Verwaltung der Klassen aus. Liegen die Klassen dagegen verteilt auf vielen Servern an beliebigen Orten, dann entspricht die Suche nach einer Klasse der Suche des Servers, der die Klasse speichert. Unter Berücksichtigung obiger Randbedingungen wurde das Problem der Klassensuche in zwei unterschiedliche Aufgaben getrennt:

1. In einen Basismechanismus, der bei der Suche nach einer Klasse garantiert das korrekte Ergebnis liefert.
2. In einen schnellen Dienst, der das Laufzeitverhalten des Agentensystems verbessern soll. Der Dienst kann manche Klassen schnell auffinden, liefert aber nicht für alle Klassen ein Ergebnis.

Während der Basismechanismus das korrekte Funktionieren der Codemigration sicherstellt, sorgt der schnelle Dienst für kürzere Wartezeiten und weniger nicht-lokale Nachrichten.

Alternative Basismechanismen wurden unter Berücksichtigung des Nachrichtenaufkommens, der Skalierbarkeit, des Fehlerverhaltens und des Ressourcenverbrauchs untersucht und bewertet. Als besonders einfach und effizient erwies sich die Erweiterung der Klassennamen um die Adresse des Servers, ein zusätzliches Konzept erlaubt sogar die Verwaltung von Replikaten.

Der schnelle Dienst beinhaltet Ersetzungsstrategien für einen begrenzen Zwischenspeicher, die Verwaltung von Nachbarserver und einen Mechanismus zur Vorhersage benötigter Klassen.

Für einen erfolgreichen Einsatz des Agentensystems ist die Gewährleistung der Sicherheit der Agenten eine Voraussetzung. Das Agentensystem wurde als sicher vorausgesetzt, darauf konnte die sichere Verbreitung der Agentenklassen aufgebaut werden.

Die Codemigration kann einfach realisiert werden, indem sie parallel zur Migration des Datenzustandes erfolgt. Optimierungen, wie sie in dieser Arbeit vorgeschlagen wurden, basieren immer auf vorausgesetzten Verhaltensweisen der Agenten, zum Beispiel vertraut die Vorratshaltung darauf, daß Agenten häufig die gleichen Klassen referenzieren. Eine Optimierung kann folglich nicht pauschal für alle Fälle erfolgen, aber durch das Sammeln von statistischen Daten wird bereits auf sich ändernde Verhaltensweisen reagiert. Haben sich gewisse Anwendungen für Agenten bewährt, dann lohnen sich Messungen am laufenden Agentensystem, um weiteres Potential zur Verbesserung der Laufzeit zu entdecken.

Die Sicherheitsmechanismen können in Zukunft weiter ausgebaut werden. Das Löschen von Klassen könnte durch einen Sicherheitsmechanismus geschützt werden, damit nicht jeder beliebige Klassen aus der Codebasis löschen kann. Mit einem sicheren Mechanismus zum automatischen Austausch von Standardklassen könnte das Agentensystem jederzeit dynamisch erweitert werden.

9 Anhang

9.1 Implementierung

Teil dieser Arbeit ist die Implementierung des Codeservers, des Klassenladers im Agentensystem und eines Werkzeugs zum komfortablen Registrieren neuer Klassen.

Im Gegensatz zu den beiden anderen Teilen ist der Codeserver sehr umfangreich. An dieser Stelle wird daher eine Teilung des Codeservers in kleinere Einheiten beschrieben. Die Sprache Java läßt nur objektorientierte Entwürfe zu, so daß eine Unterteilung in Module im Programm nicht nachvollziehbar ist. Dennoch können Gruppen von Objekten sinnvoll zu Modulen zusammengefaßt werden. Möglichst wenige Objekte, im Idealfall nur ein Objekt, stellen die Funktionalität des gesamten Moduls nach außen zur Verfügung. Die Schnittstellen, welche in der Arbeit beschrieben wurden, finden sich in Form eines Objekts (das über remote methode invocation benutzt werden kann) wieder. Diese Schnittstellenobjekte greifen auf die Schnittstellenobjekte anderer Module zurück.

Die Abbildung 20 zeigt die Zusammenhänge der Module. Die Pfeile zeigen an, welche Module auf andere Module zurückgreifen; Die Rechtecke der Abbildung stellen die Module dar. Ein Modul wird durch eine Menge von Objekten realisiert.

Die allgemeinen Dienste stellen Funktionalitäten zur Verfügung, die von allen Modulen genutzt werden. Die lokale Klassenverwaltung sorgt für die Speicherung und Rückgewinnung der Klassen im lokalen Dateisystem. Der Basismechanismus und der Zwischenspeicher erhalten jeweils getrennte Speicherkontingente zugewiesen. Die Kontingente können dadurch getrennt voneinander bemessen und verbraucht werden, außerdem wird der Zugriff auf die Codebasis, zum Beispiel für Sicherungskopien, erleichtert.

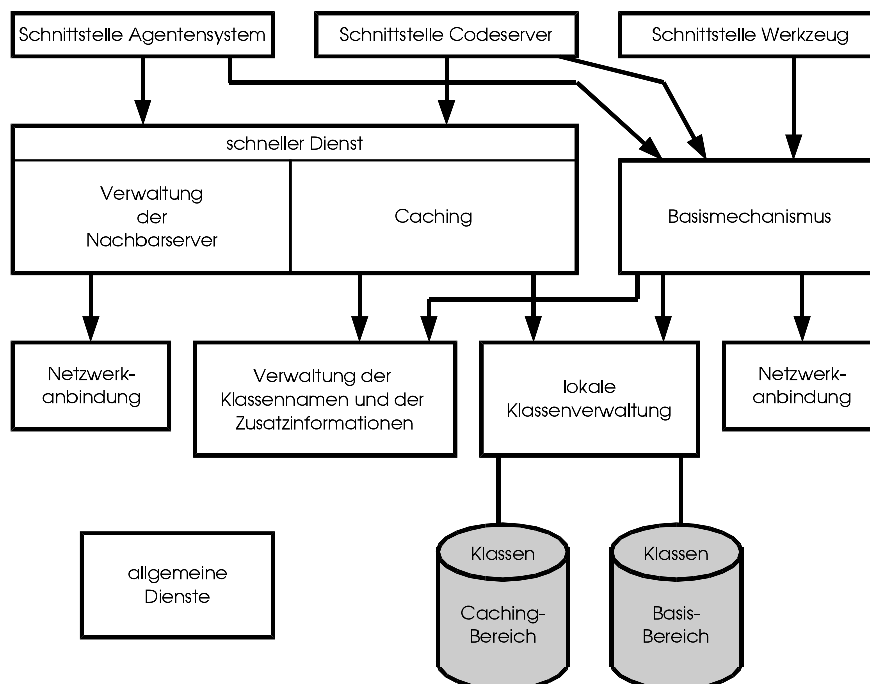

Abbildung 20: Architektur des Codeservers

Die Verwaltung der Klassennamen und der Zusatzinformationen stellt möglichst effizient Informationen über lokal vorhandene Klassen bereit. Die Speicherung der Sekundärserver und das Führen und Aktualisieren der Zugriffsstatistik werden ebenfalls von diesem Modul durchgeführt. Die Netzwerkanbindungen stellen den Kontakt zum Netzwerk her, da die Kommunikation durch die RMI gesteuert wird, entfällt eine Implementierung von speziellen Protokollen an dieser Stelle.

Der schnelle Dienst realisiert die Möglichkeiten, die in Kapitel »Verbesserung der Performanz« beschrieben wurden. Grundsätzlich sind die beiden Mechanismen: Kooperation mit Nachbarservern und Benutzung des Zwischenspeichers zu trennen. Der schnelle Dienst stellt nicht nur eine Schnittstelle zur Verfügung, sondern kontrolliert einen eigenen Thread, der regelmäßig die Verbin7dungsqualität zu den aktuellen Nachbarservern und potentiellen Nachbarservern ermittelt. Dabei können sich Veränderungen in der Reihenfolge der Nachbarserver ergeben.

Der Basismechanismus stellt eine einfache Schnittstelle bereit, allerdings sind die Anforderungen an die Funktionen hoch. Der Basismechanismus stellt die Zugriffsoperationen auf eine verteilte Datenbank zur Verfügung, die Operationen sollten möglichst zuverlässig sein. Damit sollten registrierte Klassen dauerhaft gespeichert und die referenzielle Integrität möglichst vollständig erzwungen werden. Die Einhaltung dieser Randbedingungen erfordert zahlreiche Überprüfungen, Sonder- und Fehlerbehandlungen.

Der schnelle Dienst und der Basismechanismus stellen die Grundlage dar, die zur Umsetzung der drei Schnittstellen dienen. Die Schnittstellen sind relativ einfache Objekte, welche die Funktionalität der Schnittstelle kapseln und öffentlich zugänglich machen.

Die Quelltexte sind nach den Richtlinien der Abteilung programmiert und kommentiert. Die vorgeschriebene Art der Kommentare erlaubt die automatische Erstellung einer Dokumentation der Klassen im HTML-Format. Da diese Dokumentation leichter zu aktualisieren ist, sollten die Details der Implementierung dieser Dokumentation entnommen werden.

9.2 Beispielprotokolle

Jeder Server führt über hinzugekommene und gelöschte Klassen ein Protokoll. Das folgende Beispiel soll den Zweck des Protokolls und der Merkpunkt verdeutlichen.

Die Klassen sind in diesem Beispiel mit »AAA«, »BBB«, »CCC«, usw. Bezeichnet, Merkpunkte werden mit »Mxxx« bezeichnet, wobei xxx für eine fortlaufende Nummer steht.

M057
+AAA Die Klasse AAA wurde gespeichert.
M058
+BBB
+CCC
-BBB Die Klasse BBB wurde gelöscht.
+DDD

Das Protokoll kann in folgendes äquivalente Protokoll umgeformt werden:

M057
+AAA
M058
+CCC
+DDD

Es werden weitere Klassen geladen bzw. gelöscht und es erfolgen zwei Abfragen (M059 und M060), die sich auf die Merkpunkte M057 und M058 beziehen, danach sieht das Protokoll wie folgt aus:

+AAA Die Merkpunkte M057 und M058 wurden gelöscht.
+CCC
+DDD
M059
+EEE
-DDD
+BBB

M060
-EEE
+DDD
Obwohl einige gleich Einträge mit unterschiedlichem Vorzeichen enthalten sind, können diese zur Optimierung nicht entfernt werden, da sich Merkpunkte dazwischen befinden. Lediglich die ersten drei Protokolleinträge können entfernt werden, da es keinen Merkpunkt mehr gibt, der auf diese Einträge Bezug nimmt.

Eine Anfrage mit Bezug auf M059 wird folgendermaßen beantwortet:

+EEE, -DDD, +BBB, -EEE, +DDD

Die Kürzung der Unnötigen Einträge erfordert O(n log n) Aufwand und kann durchgeführt werden. Da diese Fälle bei kurzer Zeit zwischen den Aktualisierungsabfragen nicht wahrscheinlich sind, kann dieser Aufwand eingespart werden.

Eine Anfrage mit Bezug auf Merkpunkte kleiner als 59 sollten nicht mehr auftreten, tauchen sie dennoch auf, dann wird eine Liste aller Klassen zurückgeliefert und ein neuer Merkpunkt gesetzt.

Literaturverzeichnis

[Beck 1997]
Beck, Bernhard: **Terminierung und Waisenerkennung in einem System mobiler Software-Agenten.** Diplomarbeit (noch unveröffentlicht), Fakultät Informatik, Universität Stuttgart

[Date 1991]
Date, Chris J.: **An Introduction To Database Systems.** Addison-Wesley Publishing Company, Inc. Volume I, Fifth Edition, S. 284ff.

[Fünfrocken 1996]
Fünfrocken, Stefan: **Mobile Agenten.** http://www.informatik.th-darmstadt.de/~fuenf/work/agenten/einfuehrung.html

[Gray et al. 1993]
Gray, Jim; Reuter, Andreas: **Transaction Processing - Concepts and Techniques.** Morgan Kaufmann Publishers, Inc., 1993

[Harrison et al. 1995]
Harrison, Colin G.; Chess, David M.; Kershenbaum, Aaron: **Mobile Agents: Are they a good idea?** Research Report, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY, March 1995, http://www.research.ibm.com/massive/

[Hohl 1995]
Hohl, Fritz: **Konzeption eines einfachen Agentensystems und Implementation eines Prototyps.** Diplomarbeit Nr. 1267, Fakultät Informatik, Universität Stuttgart, 1995

[Java1.1 1997]
Sun Microsystems: **JDK 1.1 New Feature Summary.** http://www.javasoft.com:80/products/jdk/1.1/docs/relnotes/features.html

[JavaAPI 1996]
Flanagan, David: **Java in a Nutshell.** O'Reilly & Associates, Inc., First Edition, February 1996, Part IV, Section 18 - Section 26

[JavaCFF 1996]
Sun Microsystems: **The Java Virtual Maschine Specification - Class File Format.** http://java.sun.com/doc/language_vm_specification.html

[JavaJCA 1997]
Sun Microsystems: **Java Cryptography Architecture API Specification & Reference.** (Stand 17. März 1997) http://java.sun.com/products/JDK/1.1/docs/guide/security/CryptoSpec.html

[JavaJCE 1997]
Sun Microsystems: **Java Cryptography Extension.** http://java.sun.com/products/JDK/1.1/docs/guide/security/index.html

[JavaRMI 1997]
Sun Microsystems: **Getting Started Using RMI.** http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/getstart.doc.html

[JavaSpec 1997]
Gosling, James; Joy, Bill; Steele, Guy: **The Java Language Specification.** Corporate and Professional Publishing Group, The Java Series, 1997

[JavaToGo 1997]
Li, Weiyi; Messerschmitt, David G.: **Java-To-Go Itinerative Computing Using Java.** http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go

[Klar 1996]
Klar, Peter: **Persistenz als Basis der Migration in einem Mobile-Agenten-System: Design und Implementierung.** Studienarbeit-Nr. 1522, Fakultät Informatik, Universität Stuttgart

[Kuhn 1997]
Kuhn, Wilfried: **Ressourcenkontrolle in einem Mobile Agentensystem.** Diplomarbeit-Nr. 1468, Fakultät Informatik, Universität Stuttgart

[Mole 1996]
Baumann, Joachim: **Mole Alpha 1.0 Documentation (1996).** http://suntrec.informatik.uni-stuttgart.de/pub/MOLE/documentation.html

[Röhrle 1996]
Röhrle, Klaus: **Finden von mobilen Agenten in einem weitverteilten System.** Studienarbeit 1539, Fakultät Informatik, Universität Stuttgart

[Straßer et al. 1996]
Straßer, Markus; Baumann, Joachim; Hohl, Fritz: **Mole - A Java Based Mobile Agent System.** Special Issues in Object-Oriented Programming, Workshop Reader of the roth European Conference on Object-Oriented Programming ECOOP '96, Linz, dpunkt.verlag 1997, S. 327 ff

[Tanenbaum 1992]
Tanenbaum, Andrew S.: **Modern Operating Systems.** Prentice-Hall International, Inc., 1992

[Terry 1985]
Terry, Douglas Brian: **Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments.** Technical Report, Xerox Corporation, Palo Alto Research Center, February 1985

[Voigt 1996]
Voigt, Thiemo: **Entwicklung und Implementation eines Modells zur quantitativen Beurteilung der Implementation und der Anwendung von Remote-Execution-Mechanismen.** Diplomarbeit-Nr. 1436, Fakultät Informatik, Universität Stuttgart

Ich versichere, daß ich diese Arbeit selbständig verfaßt
und nur die angegebenen Hilfsmittel verwendet habe.

Stuttgart, 4. April 1997